

Содержание

1	centroids	1
1.1	centroid_decomposition.cpp	1
2	fft	1
2.1	fft_advanced_integer.h	1
2.2	fft_double.h	2
2.3	fft_integer.h	3
2.4	fft_mod_10_9_7.h	4
3	flows	4
3.1	dinic.h	4
3.2	hungarian.cpp	5
3.3	min_cost_bellman_queue.h	5
3.4	min_cost_dijkstra.h	5
3.5	min_cost_ford_bellman.h	6
3.6	min_cost_negative_cycles.h	7
4	geometry	8
4.1	basic_geom.cpp	8
4.2	cutting.cpp	11
4.3	halfplane_intersection.cpp	11
4.4	point_in_poly.cpp	12
5	graphs	12
5.1	components.cpp	12
5.2	directed_mst.cpp	13
5.3	dominator_tree.h	14
5.4	edmonds_matching.h	15
5.5	euler_cycle.h	16
6	maths	16
6.1	berlekamp.h	16
6.2	crt.h	16
6.3	gauss_bitset_inverse.h	16
6.4	gauss_bitset_solve_slv.h	17
6.5	gauss_double_inverse.h	17
6.6	gauss_double_solve_slv.h	17
6.7	milller_rabin_test.h	18
7	misc	18
7.1	ch_trick_with_binary_summation_struct.cpp	18
7.2	cht_stl.cpp	18
7.3	tree_bidirectional_dp.h	19
7.4	tree_order_statistics.cpp	19
8	numeric	20
8.1	integration.cpp	20
8.2	simplex.cpp	20
9	strings	21
9.1	aho_corasick.h	21
9.2	manacher.h	21
9.3	pal_tree.h	21
9.4	prefix_function.h	22
9.5	suffix_array.cpp	22
9.6	suffix_automaton_kostroma.h	23
9.7	suffix_tree_from_automaton.cpp	24
9.8	z_function.h	24
10	templates	24
10.1	template.cpp	24
11	treap	24
11.1	treap.cpp	24

1 centroids

1.1 centroid_decomposition.cpp

```
vector<vector<int>> g;
vector<int> cnt, max_cnt;
vector<int> comp;

void dfs1(int v, int p) {
    cnt[v] = 1;
    max_cnt[v] = 0;
    comp.push_back(v);
    for (int to : g[v]) {
        if (to == p || used[to]) continue;
        dfs1(to, v);
        max_cnt[v] = max(max_cnt[v], cnt[to]);
    }
}
```

```
        cnt[v] += cnt[to];
    }

void kill_center(int v, int depth) {
    if (used[v]) {
        return;
    }
    comp.clear();
    dfs1(v, v);
    int center = -1;
    for (int x : comp) {
        if (max_cnt[x] <= cnt[v] / 2 && cnt[v] -
            cnt[x] <= cnt[v] / 2) {
            center = x;
            break;
        }
    }
    assert(center != -1);
    v = center;
    // perform actions with center v
    used[v] = true;
    for (int to : g[v]) {
        kill_center(to, depth + 1);
    }
}

void solve(__attribute__((unused)) bool read) {
    int n;
    cin >> n;

    used.assign(n, false);
    cnt.assign(n, 0);
    max_cnt.assign(n, 0);
    kill_center(0, 0);
}
```

2 fft

2.1 fft_advanced_integer.h

```
Poly derivative(Poly a) {
    if (a.empty()) {
        return a;
    }
    for (int i = 0; i < (int)a.size(); ++i) {
        a[i] = a[i] * i % mod;
    }
    a.erase(a.begin());
    return a;
}

// returns  $b(x) = \int_0^x a(t) dt$ 
Poly primitive(Poly a) {
    if (a.empty()) {
        return a;
    }
    for (int i = 0; i < (int)a.size(); ++i) {
        a[i] = a[i] * pw(i + 1, mod - 2) % mod;
    }
    a.insert(a.begin(), 0);
    return a;
}

Poly add(Poly a, const Poly& b) {
    a.resize(max(a.size(), b.size()));
    for (int i = 0; i < (int)b.size(); ++i) {
        a[i] = (a[i] + b[i]) % mod;
    }
    return a;
}

Poly sub(Poly a, const Poly& b) {
    a.resize(max(a.size(), b.size()));
    for (int i = 0; i < (int)b.size(); ++i) {
        a[i] = (a[i] + mod - b[i]) % mod;
    }
    return a;
}

Poly normalize(Poly a) {
    for (int i = 0; i < (int)a.size(); ++i) {
        a[i] = a[i] * inv[i] % mod;
    }
    return a;
}
```

```

while (!a.empty() && a.back() == 0) {
    a.pop_back();
}
return a;
}

// get such b that a · b = 1 (mod xprec)
Poly getInversed(Poly a, int prec) {
    assert(a[0]);

    Poly res = {pw(a[0], mod - 2)};
    int k = 1;
    while (k < prec) {
        k *= 2;
        Poly tmp = multiply(res, Poly({a.begin(),
            ↪ a.begin() + min(k, (int)a.size())}));
        for (auto& x : tmp) {
            x = x ? mod - x : 0;
        }
        tmp[0] = (tmp[0] + 2) % mod;

        res = multiply(tmp, res);
        res.resize(k);
    }
    res.resize(prec);
    return res;
}

// get such q and r that a = b * q + r, deg(r) < deg(b)
pair<Poly, Poly> divMod(Poly a, Poly b) {
    int n = a.size();
    int m = b.size();
    if (n < m) {
        return {{0}, a};
    }
    reverse(all(a));
    reverse(all(b));
    auto quotient = multiply(a, getInversed(b, n - m
    ↪ + 1));
    quotient.resize(n - m + 1);
    reverse(all(a));
    reverse(all(b));
    reverse(all(quotient));
    auto remainder = sub(a, multiply(b, quotient));
    while (!remainder.empty() && remainder.back() ==
    ↪ 0) {
        remainder.pop_back();
    }
    return {quotient, remainder};
}

// this is for multipoint and interpolate functions
vector<Poly> getSegmentProducts(const vector<long
    ↪ long>& pts) {
    vector<Poly> segment_polys;
    function<int(int, int)> fill_polys = [&](int l,
    ↪ int r) {
        if (l + 1 == r) {
            segment_polys.push_back({(mod - pts[l])
            ↪ % mod, 1});
            return (int)segment_polys.size() - 1;
        }
        int m = (l + r) / 2;
        int i = fill_polys(l, m);
        int j = fill_polys(m, r);
        auto new_poly = multiply(segment_polys[i],
            ↪ segment_polys[j]);
        segment_polys.push_back(new_poly);
        return (int)segment_polys.size() - 1;
    };
    fill_polys(0, pts.size());

    return segment_polys;
}

// get p and {x1, x2, ..., xn}, return {p(x1),
    ↪ p(x2), ..., p(xn)}
vector<long long> multipoint(const Poly& poly, const
    ↪ vector<long long>& pts) {
    if (pts.empty()) {
        return {};
    }

    vector<Poly> segment_polys =
    ↪ getSegmentProducts(pts);
    vector<long long> ans;
    function<void(const Poly&)> fill_ans = [&](const
    ↪ Poly& p) {
        if ((int)segment_polys.back().size() <= 2) {
            ans.push_back(p.empty() ? 0 : p[0]);
            segment_polys.pop_back();
            return;
        }
        segment_polys.pop_back();
        fill_ans(divMod(p,
            ↪ segment_polys.back()).second);
        fill_ans(divMod(p,
            ↪ segment_polys.back()).second);
    };
    fill_ans(poly);
    reverse(all(ans));

    return ans;
}

// get {x1, ..., xn} and {y1, ..., yn}, return such
    ↪ p that p(xi) = yi
Poly interpolate(const vector<long long>& xs, const
    ↪ vector<long long>& ys) {
    assert(xs.size() == ys.size());
    if (xs.empty()) {
        return {0};
    }

    vector<Poly> segment_polys = getSegmentProducts(xs);
    auto der = derivative(segment_polys.back());
    auto coeffs = multipoint(der, xs);
    for (auto& c : coeffs) {
        c = pw(c, mod - 2);
    }
    for (int i = 0; i < (int)ys.size(); ++i) {
        coeffs[i] = coeffs[i] * ys[i] % mod;
    }

    function<Poly()> get_ans = [&]() {
        Poly res;
        if (segment_polys.back().size() <= 2) {
            segment_polys.pop_back();
            res = {coeffs.back()};
            coeffs.pop_back();
        } else {
            segment_polys.pop_back();

            auto p1 = segment_polys.back();
            auto q1 = get_ans();

            auto p2 = segment_polys.back();
            auto q2 = get_ans();

            res = add(multiply(p1, q2), multiply(p2,
            ↪ q1));
        }
        return res;
    };
    return normalize(get_ans());
}

// takes 1 + b, returns b - b2/2 + b3/3 - ... mod
    ↪ x{prec}
// ofc b must be divisible by x
Poly logarithm(Poly a, int prec) {
    assert(a[0] == 1);
    auto res = primitive(multiply(derivative(a),
    ↪ getInversed(a, prec)));
    res.resize(prec);
    return res;
}

// returns 1 + a + a2/2 + a3/6 + ... mod x{prec}
// ofc a must be divisible by x
Poly exponent(Poly a, int prec) {
    assert(a[0] == 0);

    Poly res = {1};
    int k = 1;
    while (k < prec) {

```

```

    k *= 2;
    Poly tmp = {a.begin(), a.begin() + min(k,
        ↪ (int)a.size())};
    tmp[0] += 1;
    tmp = sub(tmp, logarithm(res, k));

    res = multiply(tmp, res);
    res.resize(k);
}
res.resize(prec);
return res;
}

```

2.2 fft_double.h

```

const int L = 22;
const int N = 1 << L;
bool fft_initialized = false;

using ld = long double;
using base = complex<ld>;
using Poly = vector<ld>;

const ld pi = acosl(-1);
base angles[N + 1];
int bitrev[N];

// don't know why such eps, may be changed
const ld eps = 1e-7;

inline bool eq(ld x, ld y) {
    return abs(x - y) < eps;
}

void fft_init() {
    for (int i = 0; i <= N; ++i) {
        angles[i] = {cosl(2 * pi * i / N), sinl(2 *
            ↪ pi * i / N)};
    }

    for (int i = 0; i < N; ++i) {
        int x = i;
        for (int j = 0; j < L; ++j) {
            bitrev[i] = (bitrev[i] << 1) | (x & 1);
            x >>= 1;
        }
    }

    fft_initialized = true;
}

inline int revBit(int x, int len) {
    return bitrev[x] >> (L - len);
}

void fft(vector<base>& a, bool inverse = false) {
    assert(fft_initialized &&
        ↪ "you fucking cunt just write fft_init()");
    int n = a.size();
    assert(!(n & (n - 1))); // work only with
        ↪ powers of two
    int l = __builtin_ctz(n);

    for (int i = 0; i < n; ++i) {
        int j = revBit(i, l);
        if (i < j) {
            swap(a[i], a[j]);
        }
    }

    for (int len = 1; len < n; len *= 2) {
        for (int start = 0; start < n; start += 2 *
            ↪ len) {
            for (int i = 0; i < len; ++i) {
                base x = a[start + i], y = a[start +
                    ↪ len + i];
                int idx = N / 2 / len * i;
                base w = y * angles[inverse ? N -
                    ↪ idx : idx];
                a[start + i] = x + w;
                a[start + len + i] = x - w;
            }
        }
    }
}

```

```

}

if (inverse) {
    for (auto& x : a) {
        x /= n;
    }
}

Poly multiply(Poly a, Poly b) {
    int n = 1;
    while (n < (int)a.size() || n < (int)b.size()) {
        n *= 2;
    }
    vector<base> ar(n + n), br(n + n);
    for (int i = 0; i < (int)a.size(); ++i) {
        ar[i] = a[i];
    }
    for (int i = 0; i < (int)b.size(); ++i) {
        br[i] = b[i];
    }
    fft(ar);
    fft(br);
    for (int i = 0; i < n + n; ++i) {
        ar[i] = ar[i] * br[i];
    }
    fft(ar, true);
    while (!ar.empty() && eq(norm(ar.back()), 0)) {
        ar.pop_back();
    }
    a.resize(ar.size());
    for (int i = 0; i < (int)a.size(); ++i) {
        a[i] = real(ar[i]);
    }
    return a;
}

```

2.3 fft_integer.h

```

const int mod = 998244353;
const int L = 22; // can be 23 for 998244353
const int N = 1 << L;
bool fft_initialized = false;

using Poly = vector<long long>;

long long pw(long long a, long long b) {
    long long res = 1;
    while (b) {
        if (b & 1ll) {
            res = res * a % mod;
        }
        b >>= 1;
        a = a * a % mod;
    }
    return res;
}

int getRoot() {
    int root = 1;
    while (pw(root, 1 << L) != 1 || pw(root, 1 << (L
        ↪ - 1)) == 1) {
        ++root;
    }
    return root;
}

const int root = getRoot();

long long angles[N + 1];
int bitrev[N];

void fft_init() {
    angles[0] = 1;
    for (int i = 1; i <= N; ++i) {
        angles[i] = angles[i - 1] * root % mod;
    }

    for (int i = 0; i < N; ++i) {
        int x = i;
        for (int j = 0; j < L; ++j) {
            bitrev[i] = (bitrev[i] << 1) | (x & 1);
            x >>= 1;
        }
    }
}

```

```

    }
    fft_initialized = true;
}

inline int revBit(int x, int len) {
    return bitrev[x] >> (L - len);
}

void fft(vector<long long>& a, bool inverse = false) {
    assert(fft_initialized &&
    ↪ "you fucking cunt just write fft_init()");
    int n = a.size();
    assert(!(n & (n - 1))); // work only with
    ↪ powers of two
    int l = __builtin_ctz(n);

    for (int i = 0; i < n; ++i) {
        int j = revBit(i, l);
        if (i < j) {
            swap(a[i], a[j]);
        }
    }

    for (int len = 1; len < n; len *= 2) {
        for (int start = 0; start < n; start += 2 *
        ↪ len) {
            for (int i = 0; i < len; ++i) {
                long long x = a[start + i], y =
                ↪ a[start + len + i];
                int idx = N / 2 / len * i;
                long long w = angles[inverse ? N -
                ↪ idx : idx];
                w = w * y % mod;
                a[start + i] = x + w;
                if (a[start + i] >= mod) {
                    a[start + i] -= mod;
                }
                a[start + len + i] = x - w;
                if (a[start + len + i] < 0) {
                    a[start + len + i] += mod;
                }
            }
        }

        if (inverse) {
            int rev_deg = 1;
            for (int i = 0; i < l; ++i) {
                rev_deg = (rev_deg % 2) ? ((rev_deg +
                ↪ mod) / 2) : (rev_deg / 2);
            }
            for (auto& x : a) {
                x = x * rev_deg % mod;
            }
        }
    }

    Poly multiply(Poly a, Poly b) {
        int n = 1;
        while (n < (int)a.size() || n < (int)b.size()) {
            n *= 2;
        }
        a.resize(n + n);
        b.resize(n + n);
        fft(a);
        fft(b);
        for (int i = 0; i < n + n; ++i) {
            a[i] = a[i] * b[i] % mod;
        }
        fft(a, true);
        while (!a.empty() && a.back() == 0) {
            a.pop_back();
        }
        return a;
    }
}

```

2.4 fft_mod_10_9_7.h

```

Poly multiply(const Poly& a, const Poly& b) {
    ....
    for (int i = 0; i < n; ++i) {
        answer[i] = (1i)(res[i].real() + 0.5);
    }
}

```

```

        answer[i] %= mod;
    }
    return answer;
}

const int shift = 15;
const int first_mod = 1 << shift;

Poly large_part(const Poly& a) {
    Poly res(a.size());
    for (int i = 0; i < a.size(); ++i) {
        res[i] = a[i] >> shift;
    }
    return res;
}

Poly small_part(const Poly& a) {
    Poly res(a.size());
    for (int i = 0; i < a.size(); ++i) {
        res[i] = a[i] & (first_mod - 1);
    }
    return res;
}

Poly add(const Poly& q, const Poly& w) {
    auto res = q;
    res.resize(max(q.size(), w.size()));
    for (int i = 0; i < w.size(); ++i) {
        res[i] += w[i];
    }
    return res;
}

Poly multiply_large(const Poly& a, const Poly& b,
    ↪ int k) {
    Poly largeA = large_part(a), largeB = large_part(b);
    Poly smallA = small_part(a), smallB = small_part(b);
    Poly large_mult = multiply(largeA, largeB);
    Poly small_mult = multiply(smallA, smallB);
    Poly middle_mult = multiply(add(smallA, largeA),
    ↪ add(smallB, largeB));

    Poly result(large_mult.size());
    for (int i = 0; i < result.size(); ++i) {
        result[i] = ((large_mult[i] * first_mod) %
        ↪ mod * first_mod + small_mult[i] +
        ↪ first_mod * (middle_mult[i] -
        ↪ large_mult[i] -
        ↪ small_mult[i]) % mod) %
        ↪ mod;
    }
    if (result.size() > k + 1) {
        result.resize(k + 1);
    }
    return result;
}

```

3 flows

3.1 dinic.h

```

struct Edge {
    int from, to, cap, flow;
};

const int INF = (int)2e9;

struct Dinic {
    int n;
    vector<Edge> edges;
    vector<vector<int>> g;

    Dinic(int n) : n(n) {
        g.resize(n);
    }

    void add_edge(int from, int to, int cap) {
        Edge e = {from, to, cap, 0};
        g[from].push_back(edges.size());
        edges.push_back(e);
        e = {to, from, 0, 0};
        g[to].push_back(edges.size());
    }
}

```

```

    edges.push_back(e);
}

vector<int> d;

bool bfs(int s, int t) {
    d.assign(n, INF);
    d[s] = 0;
    queue<int> q;
    q.push(s);
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (auto id : g[v]) {
            auto e = edges[id];
            if (e.cap > e.flow && d[e.to] == INF) {
                d[e.to] = d[v] + 1;
                q.push(e.to);
            }
        }
    }
    return d[t] != INF;
}

vector<int> pointer;

int dfs(int v, int t, int flow_add) {
    if (!flow_add) {
        return 0;
    }
    if (v == t) {
        return flow_add;
    }
    int added_flow = 0;
    for (int& i = pointer[v]; i < g[v].size(); ++i) {
        int id = g[v][i];
        int to = edges[id].to;
        if (d[to] != d[v] + 1) {
            continue;
        }
        int pushed = dfs(to, t, min(flow_add,
            edges[id].cap - edges[id].flow));
        if (pushed) {
            edges[id].flow += pushed;
            edges[id ^ 1].flow -= pushed;
            return pushed;
        }
    }
    return 0;
}

int max_flow(int s, int t) {
    int flow = 0;
    while (bfs(s, t)) {
        pointer.assign(n, 0);
        while (int pushed = dfs(s, t, INF)) {
            flow += pushed;
        }
    }
    return flow;
}
};

```

3.2 hungarian.cpp

```

vector<int> u(n + 1), v(m + 1), p(m + 1), way(m + 1);
for (int i = 1; i <= n; ++i) {
    p[0] = i;
    int j0 = 0;
    vector<int> minv(m + 1, INF);
    vector<char> used(m + 1, false);
    do {
        used[j0] = true;
        int i0 = p[j0], delta = INF, j1;
        for (int j = 1; j <= m; ++j) {
            if (!used[j]) {
                int cur = a[i0][j] - u[i0] - v[j];
                if (cur < minv[j]) {
                    minv[j] = cur;
                    way[j] = j0;
                }
            }
            if (minv[j] < delta) {

```

```

                delta = minv[j];
                j1 = j;
            }
        }
    } while (minv[j1] < delta);
    for (int j = 0; j <= m; ++j) {
        if (used[j]) {
            u[p[j]] += delta;
            v[j] -= delta;
        }
        else {
            minv[j] -= delta;
        }
    }
    j0 = j1;
    while (p[j0] != 0) {
        do {
            int j1 = way[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while (j0);
    }
    vector<int> ans(n + 1);
    for (int j = 1; j <= m; ++j) {
        ans[p[j]] = j;
    }
    int cost = -v[0];
}

```

3.3 min_cost_bellman_queue.h

```

using cost_type = li;
const cost_type COST_INF = (int)1e18;
const int FLOW_INF = (int)1e9;

struct MinCost {
    explicit MinCost(int n) {
        g.resize(n);
    }

    struct edge {
        int from, to;
        int cap;
        cost_type cost;
        int flow;
    };

    vector<edge> edges;
    vector<vector<int>> g;

    void add_edge(int from, int to, cost_type cost,
        int cap) {
        edge e = {from, to, cap, cost, 0};
        g[from].push_back(edges.size());
        edges.push_back(e);
        edge e2 = {to, from, 0, -cost, 0};
        g[to].push_back(edges.size());
        edges.push_back(e2);
    }

    pair<int, cost_type> min_cost(int n, int s, int
        t, bool need_max_flow, int max_flow_value =
        FLOW_INF) {
        cost_type cost = 0;
        int flow = 0;
        while (flow < max_flow_value) {
            queue<int> q;
            q.push(s);
            vector<int> in_q(n, 0);
            in_q[s] = 1;
            vector<int> p(n, -1);
            vector<cost_type> d(n);
            d[s] = 0;
            p[s] = s;
            while (!q.empty()) {
                int v = q.front();
                q.pop();
                in_q[v] = false;
                for (size_t i : g[v]) {
                    edge& e = edges[i];
                    if (e.cap == e.flow || p[e.from]
                        == -1) {
                        continue;
                    }
                    if (p[e.to] == -1 || d[e.to] >
                        d[e.from] + e.cost) {

```

```

while (changed) {
    changed = false;
    for (size_t i = 0; i < edges.size(); ++i) {
        edge &e = edges[i];
        if (e.cap == e.flow || p[e.from] == -1) continue;
        if (p[e.to] == -1 || d[e.to] > d[e.from] + e.cost) {
            d[e.to] = d[e.from] + e.cost;
            p[e.to] = i;
            changed = true;
        }
    }
}
potential = std::move(d);
}

while (flow < max_flow_value) {
    vector<cost_type> d(n);
    vector<int> p(n, -1);

    using queue_type = pair<cost_type, int>;
    priority_queue<queue_type, vector<queue_type>, greater<queue_type>> q;

    q.push({0, s});

    while (!q.empty()) {
        int v = q.top().second;
        cost_type oldD = q.top().first;
        q.pop();
        if (oldD != d[v]) continue;
        for (int id: g[v]) {
            edge &e = edges[id];
            if (e.to == s) continue;
            if (e.cap > e.flow) {
                cost_type newd = d[v] + e.cost;
                if (p[e.to] == -1 || d[e.to] > newd) {
                    d[e.to] = newd;
                    p[e.to] = id;
                    q.push({d[e.to], e.to});
                }
            }
        }
    }

    if (p[t] == -1) break;

    if (d[t] + potential[t] >= 0 && !need_max_flow) break;

    int cur = t;
    int maxAdd = max_flow_value - flow;
    while (cur != s) {
        edge &e = edges[p[cur]];
        cur = e.from;
        maxAdd = min(maxAdd, e.cap - e.flow);
    }

    flow += maxAdd;
    cost += (potential[t] + d[t]) * maxAdd;
    cur = t;
    while (cur != s) {
        int id = p[cur];
        edges[id].flow += maxAdd;
        edges[id ^ 1].flow -= maxAdd;
        cur = edges[id].from;
    }

    for (int i = 0; i < n; ++i) {
        if (i != s && p[i] == -1) {

```



```

        potential[i] = COST_INF;
    } else
        potential[i] = min(potential[i]
        ↪ + d[i], COST_INF);
    }
}

return make_pair(flow, cost);
};

```

3.5 min_cost_ford_bellman.h

```

using cost_type = li;
const cost_type COST_INF = (int)1e18;
const int FLOW_INF = (int)1e9;

struct MinCost {
    explicit MinCost(int n) {
        g.resize(n);
    }

    struct edge {
        int from, to;
        int cap;
        cost_type cost;
        int flow;
    };

    vector<edge> edges;
    vector<vector<int>>> g;

    void add_edge(int from, int to, cost_type cost,
    ↪ int cap) {
        edge e = {from, to, cap, cost, 0};
        g[from].push_back(edges.size());
        edges.push_back(e);
        edge e2 = {to, from, 0, -cost, 0};
        g[to].push_back(edges.size());
        edges.push_back(e2);
    }

    pair<int, cost_type> min_cost(int n, int s, int
    ↪ t, bool need_max_flow, int max_flow_value =
    ↪ FLOW_INF) {
        cost_type cost = 0;
        int flow = 0;
        while(flow < max_flow_value) {
            vector<int> p(n, -1);
            vector<cost_type> d(n);
            d[s] = 0;
            p[s] = s;
            bool changed = true;
            while(changed) {
                changed = false;
                for(size_t i = 0; i < edges.size();
                ↪ ++i) {
                    edge& e = edges[i];
                    if(e.cap == e.flow || p[e.from]
                    ↪ == -1)
                        continue;
                    if(p[e.to] == -1 || d[e.to] >
                    ↪ d[e.from] + e.cost) {
                        d[e.to] = d[e.from] + e.cost;
                        p[e.to] = i;
                        changed = true;
                    }
                }
            }
            if(p[t] == -1)
                break;

            if(d[t] >= 0 && !need_max_flow) {
                break;
            }

            int cur = t;
            int maxAdd = max_flow_value - flow;
            while(cur != s) {
                edge& e = edges[p[cur]];
                cur = e.from;
                maxAdd = min(maxAdd, e.cap - e.flow);
            }

```

```

        flow += maxAdd;
        cost += d[t] * maxAdd;
        cur = t;
        while(cur != s) {
            int id = p[cur];
            edges[id].flow += maxAdd;
            edges[id ^ 1].flow -= maxAdd;
            cur = edges[id].from;
        }
    }

    return make_pair(flow, cost);
};

```

3.6 min_cost_negative_cycles.h

```

using cost_type = int;
const cost_type COST_INF = (cost_type)1e9;
const int FLOW_INF = (int)1e9;

struct MinCost {
    explicit MinCost(int n) {
        g.resize(n);
    }

    struct edge {
        int from, to;
        int cap;
        cost_type cost;
        int flow;
    };

    vector<edge> edges;
    vector<vector<int>>> g;

    void add_edge(int from, int to, cost_type
    ↪ cur_cost, int cap) {
        edge e = {from, to, cap, cur_cost, 0};
        g[from].push_back(edges.size());
        edges.push_back(e);
        edge e2 = {to, from, 0, -cur_cost, 0};
        g[to].push_back(edges.size());
        edges.push_back(e2);
    }

    pair<int, cost_type> min_cost(int n, int s, int
    ↪ t, int max_flow_value = FLOW_INF) {
        cost_type cost = 0;
        int flow = 0;

        vector<int> p(n);
        vector<cost_type> d(n, 0);
        vector<int> to_add;
        while (flow < max_flow_value) {
            p.assign(n, -1);
            d.assign(n, COST_INF);
            d[s] = 0;
            set<pair<cost_type, int>> q;
            q.insert({0, s});
            vector<char> used(n, false);
            while (!q.empty()) {
                int v = q.begin()->second;
                q.erase(q.begin());
                used[v] = true;
                for (int i : g[v]) {
                    auto& e = edges[i];
                    if (e.cap == e.flow || used[e.to]) {
                        continue;
                    }
                    cost_type new_d = d[v] + e.cost;
                    if (d[e.to] > new_d) {
                        q.erase({d[e.to], e.to});
                        d[e.to] = new_d;
                        q.insert({d[e.to], e.to});
                        p[e.to] = i;
                    }
                }
            }
            if (p[t] == -1) {
                return {-1, 0};
            }
            int add_flow = max_flow_value - flow;

```

```

int cur = t;
to_add.clear();
int add_cost = 0;
while (cur != s) {
    auto& e = edges[p[cur]];
    add_flow = min(add_flow, e.cap -
    ↪ e.flow);
    to_add.push_back(p[cur]);
    cur = e.from;
    add_cost += e.cost;
}
assert(add_flow > 0);
flow += add_flow;
cost += add_flow * add_cost;
for (int x : to_add) {
    edges[x].flow += add_flow;
    edges[x ^ 1].flow -= add_flow;
}
}

int TIMER = 0;
vector<int> used_timer(n, 0);
vector<char> used(n, false);
vector<int> cur_edges;
vector<int> edges_to_add;
while (true) {
    p.assign(n, -1);
    d.assign(n, COST_INF);
    bool found = false;
    int iter = 0;
    for (int st = 0; st < s; ++st) {
        if (d[st] != COST_INF) {
            continue;
        }
        ++iter;
        d[st] = 0;
        vector<int> q, new_q;
        q.push_back(st);
        for (int it = 0; it < n; ++it) {
            ++TIMER;
            int changed = -1;
            for (int v : q) {
                for (int i : g[v]) {
                    edge &e = edges[i];
                    if (e.cap == e.flow)
                        continue;
                    cost_type new_d = d[v] +
                    ↪ e.cost;
                    if (d[e.to] > new_d) {
                        d[e.to] = new_d;
                        p[e.to] = i;
                        changed = e.to;
                        if (used_timer[e.to]
                        ↪ != TIMER) {
                            used_timer[e.to]
                            ↪ = TIMER;
                        }
                        ↪ new_q.push_back(e.to);
                    }
                }
            }
            if (changed == -1) {
                break;
            }
            sort(all(new_q));
            q.swap(new_q);
            new_q.clear();
            if (d[st] < 0) {
                changed = st;
                it = n - 1;
            }
        }
        if (it == n - 1) {
            found = true;
            int bad_end = changed;
            used.assign(n, false);
            int cur = bad_end;
            cur_edges.clear();
            while (!used[cur]) {
                used[cur] = true;
                cur_edges.push_back(p[cur]);
                cur = edges[p[cur]].from;
            }
            edges_to_add.clear();

```

```

while
    ↪ (edges[cur_edges.back()].to
    ↪ != cur) {
        ↪ edges_to_add.push_back(cur_edges.back());
        cur_edges.pop_back();
    }
    ↪ edges_to_add.push_back(cur_edges.back());
int add_cost = 0, add_flow =
    ↪ FLOW_INF;
for (auto e_id : edges_to_add) {
    add_flow = min(add_flow,
    ↪ edges[e_id].cap -
    ↪ edges[e_id].flow);
    add_cost +=
    ↪ edges[e_id].cost;
}
cost += add_cost * add_flow;
assert(add_flow > 0);
assert(add_cost < 0);
for (auto e_id : edges_to_add) {
    edges[e_id].flow +=
    ↪ add_flow;
    edges[e_id ^ 1].flow -=
    ↪ add_flow;
}
}
}
}
if (!found) {
    break;
}
return make_pair(flow, cost);
};

```

4 geometry

4.1 basic_geom.cpp

```

typedef long double dbl;

constexpr dbl eps = 1e-9;
constexpr dbl PI = 2 * acos(0);

constexpr inline dbl safe_sqrt(dbl x){
    return x < 0 ? 0 : sqrt(x);
}

constexpr inline dbl safe_acos(dbl x){
    return x < -1 ? acos(-1) : (x > 1 ? acos(1) :
    ↪ acos(x));
}

constexpr inline dbl safe_asin(dbl x){
    return x < -1 ? asin(-1) : (x > 1 ? asin(1) :
    ↪ asin(x));
}

constexpr inline dbl sqr(dbl x){
    return x * x;
}

constexpr inline bool eq(dbl x, dbl y){
    return fabs(x - y) < eps;
}

constexpr inline bool gt(dbl x, dbl y){
    return x > y + eps;
}

constexpr inline bool lt(dbl x, dbl y){
    return y > x + eps;
}

constexpr inline bool ge(dbl x, dbl y){
    return !lt(x, y);
}

constexpr inline bool le(dbl x, dbl y){

```



```

    return !gt(x, y);
}

struct pt{
    dbl x, y;
    pt(){
    pt(dbl a, dbl b):x(a), y(b){}
    pt(const pt & a):x(a.x), y(a.y){}
    pt& operator = (const pt & a){x = a.x; y = a.y;
    ↪ return *this;}
    pt operator + (const pt & a) const{return pt(x +
    ↪ a.x, y + a.y);}
    pt operator - (const pt & a) const{return pt(x -
    ↪ a.x, y - a.y);}
    pt operator * (dbl a) const{return pt(x * a, y * a);}
    pt operator / (dbl a) const{assert(fabs(a) >
    ↪ eps); return pt(x / a, y / a);}
    pt& operator += (const pt & a){x += a.x; y +=
    ↪ a.y; return *this;}
    pt& operator -= (const pt & a){x -= a.x; y -=
    ↪ a.y; return *this;}
    pt& operator *= (dbl a){x *= a; y *= a; return
    ↪ *this;}
    pt& operator /= (dbl a){assert(fabs(a) > eps); x
    ↪ /= a; y /= a; return *this;}
    bool isZero() const{return fabs(x) < eps &&
    ↪ fabs(y) < eps;}
    bool operator == (const pt & a) const{return
    ↪ (*this - a).isZero();}
    bool operator != (const pt & a) const{return
    ↪ !(*this == a);}
    dbl cross(const pt & a) const{return x * a.y - y
    ↪ * a.x;}
    dbl cross(pt a, pt b) const{
    ↪ a -= *this; b -= *this;
    ↪ return a.cross(b);
    }
    dbl dot(const pt & a) const{return x * a.x + y *
    ↪ a.y;}
    dbl dot(pt a, pt b) const{
    ↪ a -= *this; b -= *this;
    ↪ return a.dot(b);
    }
    dbl length() const{return sqrt(sqr(x) + sqr(y));}
    dbl sqrLength() const{return x * x + y * y;}
    void normalizeSelf(dbl len = 1.0){*this /=
    ↪ length(); *this *= len;}
    pt normalize(dbl len = 1.0) const{
    ↪ pt res(*this);
    ↪ res.normalizeSelf(len);
    ↪ return res;
    }
    dbl dist(const pt & a) const{return (*this -
    ↪ a).length();}
    dbl angle() const{return atan2(y, x);}
    void rotateSelf(dbl phi){
    ↪ dbl pcos = cos(phi), psin = sin(phi);
    ↪ dbl nx = x * pcos - y * psin, ny = y * pcos
    ↪ ↪ + x * psin;
    ↪ x = nx; y = ny;
    }
    void rotateSelf(dbl cosphi, dbl sinphi){
    ↪ dbl nx = x * cosphi - y * sinphi, ny = y *
    ↪ ↪ cosphi + x * sinphi;
    ↪ x = nx; y = ny;
    }
    pt rotate(dbl phi) const{
    ↪ pt res(*this);
    ↪ res.rotateSelf(phi);
    ↪ return res;
    }
    pt rotate(dbl cosphi, dbl sinphi) const{
    ↪ pt res(*this);
    ↪ res.rotateSelf(cosphi, sinphi);
    ↪ return res;
    }
    void out() const{
    ↪ cout << fixed << x << " " << y << '\n';
    }
    void outf() const{
    ↪ printf("%.15lf %.15lf\n", (double)x, (double)y);
    }
};

bool lexComp(const pt & l, const pt & r){
    ↪ if(fabs(l.x - r.x) > eps){
    ↪ ↪ return l.x < r.x;
    }
    ↪ else return l.y < r.y;
    }

dbl angle(pt l, pt mid, pt r){
    ↪ l -= mid; r -= mid;
    ↪ return atan2(l.cross(r), l.dot(r));
    }

inline pt trBary(pt a, pt b, pt c, dbl wa, dbl wb,
    ↪ dbl wc){
    ↪ return (a * wa + b * wb + c * wc)/(wa + wb + wc);
    }

inline pt trCent(pt a, pt b, pt c){
    ↪ return trBary(a, b, c, 1, 1, 1);
    }

inline pt trIncent(pt a, pt b, pt c){
    ↪ return trBary(a, b, c, (b - c).length(), (c -
    ↪ ↪ a).length(), (a - b).length());
    }

inline pt trCirc(pt a, pt b, pt c){
    ↪ dbl la = (b - c).sqrLength(), lb = (c -
    ↪ ↪ a).sqrLength(), lc = (a - b).sqrLength();
    ↪ return trBary(a, b, c, la * (lb + lc - la), lb *
    ↪ ↪ (lc + la - lb), lc * (la + lb - lc));
    }

inline pt trOrth(pt a, pt b, pt c){
    ↪ dbl la = (b - c).sqrLength(), lb = (c -
    ↪ ↪ a).sqrLength(), lc = (a - b).sqrLength();
    ↪ return trBary(a, b, c, (la + lb - lc) * (la + lc
    ↪ ↪ - lb), (lb + la - lc) * (lb + lc - la), (lc
    ↪ ↪ + la - lb) * (lc + lb - la));
    }

inline pt trExc(pt a, pt b, pt c){
    ↪ dbl la = (b - c).length(), lb = (c -
    ↪ ↪ a).length(), lc = (a - b).length();
    ↪ return trBary(a, b, c, -la, lb, lc);
    }

struct Line{
    pt p[2];
    dbl a, b, c;
    Line(){
    ↪ void recalcEquation(){
    ↪ ↪ a = p[1].y - p[0].y;
    ↪ ↪ b = p[0].x - p[1].x;
    ↪ ↪ c = -a * p[0].x - b * p[0].y;
    }
    ↪ void normalizeEquation(){
    ↪ ↪ dbl norm = sqrt(sqr(a) + sqr(b));
    ↪ ↪ a /= norm; b /= norm; c /= norm;
    ↪ ↪ if(a < -eps || (fabs(a) < eps && b < -eps)){
    ↪ ↪ ↪ a = -a; b = -b; c = -c;
    }
    }
    Line(pt l, pt r){p[0] = l; p[1] = r;
    ↪ ↪ recalcEquation();}
    Line(dbl pa, dbl pb, dbl pc){
    ↪ ↪ a = pa; b = pb; c = pc;
    ↪ ↪ if(fabs(b) < eps)p[0] = pt{-c/a, 0};
    ↪ ↪ else p[0] = pt{0, -c/b};
    ↪ ↪ p[1] = pt(p[0].x - b, p[0].y + a);
    }
    pt& operator [] (const int & i){return p[i];}
    ↪ const pt& operator [] (const int & i) const{return
    ↪ ↪ p[i];}
    Line(const Line & l){
    ↪ ↪ p[0] = l.p[0]; p[1] = l.p[1];
    ↪ ↪ a = l.a; b = l.b; c = l.c;
    }
    vector<dbl> getEquation() const{return
    ↪ ↪ vector<dbl>{a, b, c};}
    vector<dbl> getNormEquation() const{

```

```

    Line tmp(*this);
    tmp.normalizeEquation();
    return tmp.getEquation();
}
pt getOrth()const{
    return pt(a, b);
}
pt getNormOrth()const{
    Line tmp(*this);
    tmp.normalizeEquation();
    return tmp.getOrth();
}
int signPoint(const pt & t)const{
    dbl val = a * t.x + b * t.y + c;
    if(val < -eps)return -1;
    if(val > eps)return 1;
    return 0;
}
bool hasPointLine(const pt & t)const{
    return signPoint(t) == 0;
}
bool hasPointSeg(const pt & t)const{
    return hasPointLine(t) && t.dot(p[0], p[1])
    ↪ < eps;
}
dbl distToPt(const pt & t)const{
    return fabs(a * t.x + b * t.y +
    ↪ c)/getOrth().length();
}
dbl distToPtSeg(const pt & t)const{
    if(le(p[0].dot(t, p[1]), 0))return p[0].dist(t);
    if(le(p[1].dot(t, p[0]), 0))return p[1].dist(t);
    return distToPt(t);
}
};

struct Circle{
    pt c;
    dbl r;
    Circle(){
    Circle(dbl x, dbl y, dbl rr):c(x, y), r(rr){}
    Circle(const pt & p, dbl rr):c(p), r(rr){}
    Circle(const Circle & x):c(x.c), r(x.r){}
    Circle& operator = (const Circle & x){
        c = x.c; r = x.r;
        return *this;
    }
    dbl area()const{return PI * sqr(r);}
    dbl diam()const{return 2 * r;}
    dbl perim()const{return diam() * PI;}
    bool operator == (const Circle & a)const{
        return c == a.c && fabs(r - a.r) < eps;
    }
    pt getByAngle(dbl ang)const{
        return c + pt(r * cos(ang), r * sin(ang));
    }
    bool hasPointCircle(const pt & p){return
    ↪ c.dist(p) < r + eps;}
    bool onPointCircle(const pt & p){return
    ↪ eq(c.dist(p), r);}
    bool inPointCircle(const pt & p){return
    ↪ hasPointCircle(p) && !onPointCircle(p);}
};

pt projPtLine(pt p, Line l){
    pt vec = l[1] - l[0];
    return l[0] + vec * (vec.dot(p -
    ↪ l[0])/vec.dot(vec));
}

pt reflectPtLine(pt p, Line l){
    pt q = projPtLine(p, l);
    return p + (q - p) * 2;
}

vector<pt> interLineLine(Line l1, Line l2){
    if(fabs(l1.getOrth().cross(l2.getOrth())) < eps){
        if(l1.hasPointLine(l2[0]))return {l1[0], l1[1]};
        else return {};
    }
    pt u = l2[1] - l2[0];
    pt v = l1[1] - l1[0];
    dbl s = u.cross(l2[0] - l1[0])/u.cross(v);

    return {pt(l1[0] + v * s)};
}

vector<pt> interSegSeg(Line l1, Line l2){
    if(l1[0] == l1[1]){
        if(l2[0] == l2[1]){
            if(l1[0] == l2[0])return {l1[0]};
            else return {};
        }
        else{
            if(l2.hasPointSeg(l1[0]))return {l1[0]};
            else return {};
        }
    }
    if(l2[0] == l2[1]){
        if(l1.hasPointSeg(l2[0]))return {l2[0]};
        else return {};
    }
    auto li = interLineLine(l1, l2);
    if(li.empty())return li;
    if(li.size() == 2){
        if(!lexComp(l1[0], l1[1]))swap(l1[0], l1[1]);
        if(!lexComp(l2[0], l2[1]))swap(l2[0], l2[1]);
        vector<pt> res(2);
        if(lexComp(l1[0], l2[0]))res[0] = l2[0];
        ↪ else res[0] = l1[0];
        if(lexComp(l1[1], l2[1]))res[1] = l1[1];
        ↪ else res[1] = l2[1];
        if(res[0] == res[1])res.pop_back();
        if((int)res.size() == 2 && lexComp(res[1],
        ↪ res[0]))return res;
        else return res;
    }
    pt cand = li[0];
    if(l1.hasPointSeg(cand) &&
    ↪ l2.hasPointSeg(cand))return {cand};
    else return {};
}

vector<pt> interLineSeg(Line l1, Line l2){
    if(abs((l1[0] - l1[1]).cross(l2[0] - l2[1])) < eps){
        if(l1.hasPointLine(l2[0])){if(lexComp(l2[1],
        ↪ l2[0])) return {l2[1], l2[0]}; else
        ↪ return {l2[0], l2[1]};}
        else return {};
    }
    pt cand = interLineLine(l1, l2)[0];
    if(l2.hasPointSeg(cand))return {cand};
    else return {};
}

vector<pt> interLineCircle(Line l, Circle c){
    dbl d = l.distToPt(c.c);
    if(d > c.r + eps)return {};
    if(fabs(d - c.r) < eps){
        return {projPtLine(c.c, l)};
    }
    pt p = projPtLine(c.c, l);
    dbl lol = safe_sqrt(sqr(c.r) - sqr(d));
    lol /= (l[1] - l[0]).length();
    return {p + (l[1] - l[0])*lol, p - (l[1] -
    ↪ l[0])*lol};
}

vector<pt> interSegCircle(Line l, Circle c){
    auto cand = interLineCircle(l, c);
    vector<pt> res;
    for(pt p :
    ↪ cand)if(l.hasPointSeg(p))res.push_back(p);
    return res;
}

vector<pt> interCircleCircle(Circle c1, Circle c2){
    if(c1.r + eps < c2.r)swap(c1, c2);
    if(c1 == c2){
        return {c1.getByAngle(0),
        ↪ c1.getByAngle(PI/2), c1.getByAngle(PI)};
    }
    pt vec = c2.c - c1.c;
    dbl d = vec.length();
    dbl ang = vec.angle();
    dbl longest = max(max(c1.r, c2.r), d);
    dbl per = c1.r + c2.r + d;

```

```

    if(2 * longest > per + eps)return {};
    if(abs(2 * longest - per) < 2 * eps)return
    ↪ {c1.getByAngle(ang)};
    dbl cang = safe_acos((sqr(c1.r) + sqr(d) -
    ↪ sqr(c2.r))/(2*c1.r*d));
    return {c1.getByAngle(ang + cang),
    ↪ c1.getByAngle(ang - cang)};
}

vector<pt> tangentsPtCircle(pt p, Circle c){
    dbl d = (c.c - p).length();
    if(d < c.r - eps)return {};
    if(fabs(d - c.r) < eps)return {p};
    dbl ang = safe_acos(c.r/d);
    dbl cang = (p - c.c).angle();
    return {c.getByAngle(cang - ang),
    ↪ c.getByAngle(cang + ang)};
}

vector<Line> outerTangents(Circle c1, Circle c2){
    if(c1 == c2){return {Line(0, 0, 0)};}
    if(c1.r > c2.r)swap(c1, c2);
    dbl d = (c1.c - c2.c).length();
    if(c1.r + d < c2.r - eps)return {};
    if(fabs(c1.r - c2.r) < eps){
        dbl ang = (c2.c - c1.c).angle();
        pt l = c1.getByAngle(ang + PI/2), r =
        ↪ c1.getByAngle(ang - PI/2);
        return {{l, l + (c2.c - c1.c)}, {r, r +
        ↪ (c2.c - c1.c)}};
    }
    pt p = c2.c + (c1.c - c2.c) * (c2.r/(c2.r - c1.r));
    if(c1.r + d < c2.r + eps){
        return {{p, p + (c1.c - c2.c).rotate(PI/2)}};
    }
    dbl ang = safe_asin((c2.r - c1.r)/d);
    return {{p, p + (c1.c - p).rotate(ang)}, {p, p +
    ↪ (c1.c - p).rotate(-ang)}};
}

vector<Line> innerTangents(Circle c1, Circle c2){
    if(c1 == c2){return {};}
    if(c1.r < c2.r)swap(c1, c2);
    dbl d = (c1.c - c2.c).length();
    if(d < c1.r + c2.r - eps)return {};
    pt p = c1.c + (c2.c - c1.c) * (c1.r/(c1.r + c2.r));
    if(d < c1.r + c2.r + eps){
        return {{p, p + (c1.c - p).rotate(PI/2)}};
    }
    dbl ang = safe_acos(c1.r/(p - c1.c).length());
    dbl cang = (p - c1.c).angle();
    pt l = c1.getByAngle(cang + ang), r =
    ↪ c1.getByAngle(cang - ang);
    return {{p, l}, {p, r}};
}

vector<Line> allTangents(Circle c1, Circle c2){
    auto kek = outerTangents(c1, c2), bishkek =
    ↪ innerTangents(c1, c2);
    for(auto lol : kek)bishkek.push_back(lol);
    return bishkek;
}

```

4.2 cutting.cpp

```

vector<pt> cutConvex(Polygon p, Line ln, Polygon &
    ↪ l, Polygon & r){
    int n = p.size();
    l.clear(); r.clear();
    bool side = false;
    vector<pt> cutp;
    for(int i = 0; i < n; i++){
        int j = p.nxt(i);
        auto cand = interLineSeg(ln, {p[i], p[j]});
        if(cand.empty()){
            if(!side){l.push_back(p[j]);}
            else {r.push_back(p[j]);}
            continue;
        }
        if(cand.size() == 2){
            l = Polygon();
            r = p;

```

```

        return cand;
    }
    pt curr = cand[0];
    if(curr == p[i]){
        if(!side){l.push_back(p[i]);
        ↪ l.push_back(p[j]);} else
        ↪ {r.push_back(p[i]);
        ↪ r.push_back(p[j]);}
        continue;
    }
    if(curr == p[j]){
        cutp.push_back(p[j]);
        if(!side){l.push_back(p[j]); else
        ↪ r.push_back(p[j]);}
        side = !side;
        continue;
    }
    cutp.push_back(curr);
    if(!side){l.push_back(curr);
    ↪ r.push_back(curr); r.push_back(p[j]);}
    else {r.push_back(curr); l.push_back(curr);
    ↪ l.push_back(p[j]);}
    side = !side;
}
if(cutp.size() == 1){
    l = Polygon();
    r = p;
}
return cutp;
}

dbl cutPolygon(Polygon & p, Line l){
    int n = p.size();
    vector<pair<dbl, int>> events;
    for(int i = 0; i < n; i++){
        int j = p.nxt(i);
        int is = l.signPoint(p[i]), js =
        ↪ l.signPoint(p[j]);
        if(is == js)continue;
        dbl pos = (l[1] - l[0]).dot(interLineLine(l,
        ↪ Line(p[i], p[j]))[0] - l[0])/(l[1] -
        ↪ l[0]).length();
        if(is < js)events.push_back(make_pair(pos,
        ↪ is && js ? 2 : 1));
        else events.push_back(make_pair(pos, is &&
        ↪ js ? -2 : -1));
    }
    sort(events.begin(), events.end());
    int bal = 0;
    dbl ans = 0;
    F(i, 0, (int)events.size()){
        if(bal)ans += events[i].first - events[i -
        ↪ 1].first;
        bal += events[i].second;
    }
    return ans;
}

```

4.3 halfplane_intersection.cpp

```

using ld = double;
const ld eps = 1e-9;

struct point {
    ld x, y;

    point(ld x = 0, ld y = 0): x(x), y(y) {}

    point operator+(const point& p) const { return
    ↪ point(x + p.x, y + p.y); }
    point operator-(const point& p) const { return
    ↪ point(x - p.x, y - p.y); }

    point operator*(ld t) const { return point(x *
    ↪ t, y * t); }
    point operator/(ld t) const { return point(x /
    ↪ t, y / t); }

    point rot() const { return point(-y, x); }

    ld vprod(const point& p) const { return x * p.y
    ↪ - y * p.x; }

```

```

ld sprod(const point& p) const { return x * p.x
↪ + y * p.y; }

int half() const {
    if (y)
        return y < -eps;
    else
        return x < -eps;
}

ld sql() const { return x * x + y * y; }
ld len() const { return sqrt(sql()); }

bool operator<(const point& p) const { return
↪ make_pair(x, y) < make_pair(p.x, p.y); }
};

int sign(ld x) {
    return abs(x) > eps ? (x > 0 ? 1 : -1) : 0;
}

int vecLess(const point& a, const point& b) {
    if (a.half() != b.half())
        return a.half() < b.half() ? 1 : -1;
    else {
        return sign(a.vprod(b));
    }
}

struct halfplane {
    // ax + by + c >= 0
    ld a, b, c;
    int type;

    tuple<ld, ld, ld> get() const { return
↪ make_tuple(a, b, c); }
    bool operator<(const halfplane& rhs) const {
↪ return get() < rhs.get(); }

    point norm() const { return point(a, b); }

    point intersect(const halfplane& h) const {
        ld x = -c * h.b + b * h.c;
        ld y = a * -h.c + c * h.a;
        ld denum = a * h.b - b * h.a;
        return point(x / denum, y / denum);
    }
};

// does intersection of a and c belong to b?
// assumes that a.vprod(c) > 0!
bool interAccepted(const halfplane& a, const
↪ halfplane& b, const halfplane& c) {
    // Determinant of 3x3 matrix formed by a, b, c
    return a.a * (b.b * c.c - b.c * c.b) - a.b *
↪ (b.a * c.c - b.c * c.a) + a.c * (b.a * c.b -
↪ b.b * c.a) < 0;
}

void sanitizeHalfplanes(vector<halfplane>& planes,
↪ bool doAdd, bool doSort) {
    // Add bounding box
    const ld INF = 1e9;
    if (doAdd) {
        planes.push_back(halfplane { 1, 0, INF });
        planes.push_back(halfplane { -1, 0, INF });
        planes.push_back(halfplane { 0, 1, INF });
        planes.push_back(halfplane { 0, -1, INF });
    }

    // Normalize halfplanes. This is used when
    ↪ selecting strictest of parallel halfplanes
    // NOT NEEDED if there are no collinear (and not
    ↪ antiparallel) normals, but may improve
    ↪ precision
    for (halfplane& h: planes) {
        ld len = h.norm().len();
        h.a /= len;
        h.b /= len;
        h.c /= len;
    }

    if (doSort)
        sort(all(planes), [&](halfplane& a,
↪ halfplane& b) { return vecLess(a.norm(),
↪ b.norm()) > 0; });

    class polygon {
    public:
        vector<point> pts;

        polygon(const vector<point>& pts =
↪ vector<point>()): pts(pts) {}

        ld getDoubleSquare() const {
            ld result = 0;
            int n = pts.size();
            for (int i = 1; i < n - 1; ++i) {
                result += (pts[i] - pts[0]).vprod(pts[i]
↪ + 1] - pts[0]);
            }
            return abs(result);
        }
    };

    // Returns halfplane through points a and b,
    // inner part is counter-clockwise from a->b segment
    halfplane byPoints(point a, point b) {
        // rot counter clockwise, n points to area
        ↪ inside halfplane intersection
        point n = (b - a).rot();
        return halfplane { n.x, n.y, -n.sprod(a) };
    }

    // empty return polygon/vector denotes empty
    ↪ intersection
    // degenerate intersections are reported as empty

    // CALL sanitizeHalfplanes WITH SORT AND/OR ADD
    ↪ BOUNDING BOX BEFORE USING!
    polygon getPolygon(const vector<halfplane>& planes) {
        int l = 0, r = 0;
        static vector<halfplane> ans;
        ans.clear();
        ans.reserve(planes.size());

        for (int L = 0; L < planes.size(); ) {
            int R = L + 1;
            while (R < planes.size() &&
↪ abs(planes[L].norm().vprod(planes[R].norm()))
↪ < eps) ++R;

            // choose most powerful inequality among
            ↪ those with equal normals
            // assumes that normals are identity!
            const halfplane& h =
↪ *min_element(planes.begin() + L,
↪ planes.begin() + R, [](const halfplane&
↪ a, const halfplane& b) { return a.c <
↪ b.c; });
            L = R;

            while (r - l > 1 && !interAccepted(ans[r -
↪ 2], h, ans[r - 1])) {
                ans.pop_back();
                --r;
            }

            while (r - l > 1 && !interAccepted(ans[l],
↪ h, ans[l + 1])) {
                ++l;
            }

            // WATCH OUT: you may need to tweak eps here
            ↪ for severe problems
            if (r - l > 0 && ans[r -
↪ 1].norm().vprod(h.norm()) <= -1e-7) {
                return polygon();
            }

            if (r - l < 2 || interAccepted(ans[r - 1],
↪ ans[l], h)) {
                ans.push_back(h);
                r++;
            }
        }
    }
}

```

```

    }

    assert(r == ans.size());

    // IF YOU NEED HALFPLANES:
    // return vector<halfplane>(ans.begin() + 1,
    ↪ ans.end());

    int n = r - 1;

    polygon poly;
    poly.pts.reserve(n);
    for (int i = 0; i < n; ++i) {
        poly.pts.push_back(ans[l +
        ↪ i].intersect(ans[l + (i + 1) % n]));
    }

    return poly;
}

```

4.4 point_in_poly.cpp

```

bool insidePtPoly(const Polygon & p, pt a){
    for(int i = 0; i < (int)p.p.size(); i++){
        if(Line(p.p[i],
        ↪ p.p[p.nxt(i)]).hasPointSeg(a))return
        ↪ true;
    }
    int wn = 0;
    for(int i = 0; i < (int)p.p.size(); i++){
        int j = p.nxt(i);
        if(p.p[i].y < a.y + eps){
            if(a.y + eps < p.p[j].y){
                if(p.p[i].cross(p.p[j], a) > eps)++wn;
            }
        }
        else{
            if(p.p[j].y < a.y + eps){
                if(p.p[i].cross(p.p[j], a) < -eps)--wn;
            }
        }
    }
    return wn != 0;
}

```

5 graphs

5.1 components.cpp

```

struct Graph {
    void read() {
        int m;
        cin >> n >> m;

        e.resize(n);

        for (int i = 0; i < m; ++i) {
            int u, v;
            cin >> u >> v;
            --u; --v;
            e[u].push_back(v);
            e[v].push_back(u);
        }

        /* COMMON PART */

        int n;
        vector<vector<int>> e;

        int counter = 1;
        vector<int> inTime, minInTime;

        void dfs(int v, int p = -1) {
            minInTime[v] = inTime[v] = counter++;

            for (int u: e[v]) {
                if (u == p) continue;

                if (!inTime[u]) {
                    dfs(u, v);
                }
            }
        }
    }
}

```

```

        minInTime[v] = min(minInTime[v],
        ↪ minInTime[u]);
    }
    else {
        minInTime[v] = min(minInTime[v],
        ↪ inTime[u]);
    }
}

vector<char> used;

/* COMPONENTS SEPARATED BY BRIDGES (COLORING) */

int nColors;
vector<int> color;

void colorDfs(int v, int curColor) {
    color[v] = curColor;

    for (int u: e[v]) {
        if (color[u] != -1) continue;

        colorDfs(u, minInTime[u] > inTime[v] ?
        ↪ nColors++ : curColor);
    }
}

void findVertexComponents() {
    inTime.assign(n, 0);
    minInTime.assign(n, 0);
    counter = 1;

    for (int i = 0; i < n; ++i)
        if (!inTime[i])
            dfs(i);

    nColors = 0;
    color.assign(n, -1);
    for (int i = 0; i < n; ++i)
        if (color[i] == -1) {
            colorDfs(i, nColors++);
        }
}

/* COMPONENTS SEPARATED BY JOINTS (EDGE
↪ COMPONENTS) */

struct Edge {
    int u, v;
};

// Cactus loops can be parsed as .u of every edge
vector<vector<Edge>> edgeComps;

vector<int> colorStack;

void edgeCompDfs(int v, int p = -1) {
    used[v] = true;

    for (int u: e[v]) {
        if (used[u]) {
            if (inTime[u] < inTime[v] && u != p) {
                // NOTE: && u != p makes
                ↪ one-edge components contain
                ↪ exactly one edge;
                // if you need them as two-edge
                ↪ loops, remove this part of
                ↪ if condition

                ↪ edgeComps[colorStack.back()].push_back({v,
                ↪ u});
            }

            continue;
        }

        bool newComp = minInTime[u] >= inTime[v];

        if (newComp) {
            colorStack.push_back(edgeComps.size());
            edgeComps.emplace_back();
        }
    }
}

```

```

    ↪ edgeComps[colorStack.back()].push_back({v,
    ↪ u});
    edgeCompDfs(u, v);

    if (newComp) {
        colorStack.pop_back();
    }
}

void findEdgeComponents() {
    inTime.assign(n, 0);
    minInTime.assign(n, 0);
    counter = 1;

    for (int i = 0; i < n; ++i)
        if (!inTime[i])
            dfs(i);

    used.assign(n, false);
    colorStack.clear();
    edgeComps.clear();
    for (int i = 0; i < n; ++i)
        if (!used[i]) {
            assert(colorStack.empty());
            edgeCompDfs(i);
        }
}
};

```

5.2 directed_mst.cpp

```

vector<int> min_edges;

// RETURNS: value of directed MST with root in root
// ids of min egdes are pushed into min_edges
// WARNING: DO NOT FORGET TO FILL edge.id !!!
↪ (algorithm reports these values)
li findMst(vector<edge>& edges, int n, int root) {
    li res = 0;

    const li INF = 1e18;
    vector<li> minCost(n, INF);
    vector<int> id_edge(n, -1);

    for (int i = 0; i < edges.size(); i++)
        edges[i].local_id = i;

    for (edge& e: edges) {
        if (e.from == e.to || e.to == root) continue;

        if (minCost[e.to] > e.cost) {
            minCost[e.to] = e.cost;
            id_edge[e.to] = e.id;
        }
    }

    for (int v = 0; v < n; v++)
        if (v != root) {
            res += minCost[v];
        }

    vector<edge> zero;
    for (edge& e: edges) {
        if (e.from == e.to || e.to == root) continue;

        e.cost -= minCost[e.to];
        if (e.cost == 0)
            zero.push_back(e);
    }

    vector<vector<tuple<int, int, int>>> zero_to(n),
    ↪ zero_to_rev(n);
    for (edge& e: zero) {
        zero_to[e.from].emplace_back(e.to, e.id,
        ↪ e.local_id);
        zero_to_rev[e.to].emplace_back(e.from, e.id,
        ↪ e.local_id);
    }

    vector<char> used(n, false);
    vector<int> out_order;

```

```

    vector<int> can_min;
    function<void(int)> dfs = [&](int v) {
        used[v] = true;
        for (auto ed: zero_to[v]) {
            int u = get<0>(ed);

            if (!used[u]) {
                dfs(u);
                can_min.push_back(get<1>(ed));
            }
        }
        out_order.push_back(v);
    };

    dfs(root);

    bool fail = false;
    for (int v = 0; v < n; v++)
        if (!used[v]) {
            fail = true;
            dfs(v);
        }

    if (!fail) {
        min_edges = can_min;
        answer += res;
        return res;
    }

    reverse(all(out_order));

    vector<int> color(n, -1);

    int curColor = 0;

    function<void(int)> colorDfs = [&](int v) {
        color[v] = curColor;

        for (auto ed: zero_to_rev[v]) {
            int u = get<0>(ed);
            if (color[u] == -1) {
                colorDfs(u);
                min_edges.push_back(get<2>(ed));
            }
        }
    };

    for (int v: out_order) {
        if (color[v] == -1) {
            colorDfs(v);
            curColor++;
        }
    }

    vector<edge> new_edges;
    for (int i = 0; i < edges.size(); i++) {
        edge& e = edges[i];
        if (e.from == e.to || e.to == root) continue;

        if (color[e.to] != color[e.from]) {
            edge new_e = edge { color[e.from],
            ↪ color[e.to], e.cost };
            new_e.id = i;
            new_edges.push_back(new_e);
        }
    }

    answer += res;
    li mst_res = findMst(new_edges, curColor,
    ↪ color[root]);
    res += mst_res;

    can_min.clear();
    used.assign(n, false);

    function<void(int)> sc_dfs = [&](int v) {
        used[v] = true;
        for (auto ed: zero_to[v]) {
            int u = get<0>(ed);
            if (color[u] == color[v] && !used[u]) {
                assert(get<1>(ed) >= 0);
                min_edges.push_back(get<2>(ed));
                sc_dfs(u);
            }
        }
    };

```



```

    }
}
};

for (int i = 0; i < min_edges.size(); i++) {
    int id = min_edges[i];
    edge& e = edges[id];
    can_min.push_back(e.id);

    sc_dfs(e.to);
}

sc_dfs(root);

min_edges = can_min;
return res;
}

```

5.3 dominator_tree.h

```

struct DominatorTree {
    int n;
    int root;
    vector<int> tin, revin;
    vector<int> sdом, idом;
    vector<vector<int>> g, revg;
    vector<int> parent;

    vector<int> dsu;
    vector<int> min_v;
    int cnt = 0;

    int get(int v) {
        ++cnt;
        if (dsu[v] == v) {
            return v;
        }
        int next_v = get(dsu[v]);
        if (sdом[min_v[dsu[v]]] < sdом[min_v[v]]) {
            min_v[v] = min_v[dsu[v]];
        }
        dsu[v] = next_v;
        return next_v;
    }

    void merge(int from, int to) {
        dsu[from] = to;
    }

    DominatorTree(int n, int root): n(n),
    ↪ root(root), dsu(n) {
        tin.resize(n, -1);
        revin.resize(n, -1);
        sdом.resize(n);
        idом.resize(n);
        g.resize(n);
        revg.resize(n);
        dsu.resize(n);
        parent.assign(n, -1);
        min_v.assign(n, -1);
        for (int i = 0; i < n; ++i) {
            dsu[i] = i;
            min_v[i] = i;
            sdом[i] = i;
            idом[i] = i;
        }

        void dfs(int v, vector<vector<int>>& cur_g, int& ↪ timer) {
            tin[v] = timer++;
            for (int to : cur_g[v]) {
                if (tin[to] == -1) {
                    dfs(to, cur_g, timer);
                    parent[tin[to]] = tin[v];
                }
                revg[tin[to]].push_back(tin[v]);
            }
        }

        vector<int> get_tree(vector<vector<int>> cur_g) {
            vector<char> used(n, false);
            int timer = 0;
            dfs(root, cur_g, timer);
        }
    }
}

```

```

for (int i = 0; i < n; ++i) {
    if (tin[i] == -1) {
        continue;
    }
    revin[tin[i]] = i;
    for (int to : cur_g[i]) {
        g[tin[i]].push_back(tin[to]);
    }
}

vector<vector<int>> buckets(n);
for (int i = n - 1; i >= 0; --i) {
    for (int to : revg[i]) {
        get(to);
        sdом[i] = min(sdом[i], sdом[min_v[to]]);
    }
    if (revin[i] == -1) {
        continue;
    }
    if (i) {
        buckets[sdом[i]].push_back(i);
    }
    for (int w : buckets[i]) {
        get(w);
        int v = min_v[w];
        if (sdом[v] == sdом[w]) {
            idом[w] = sdом[w];
        } else {
            idом[w] = v;
        }
    }
    for (int to : g[i]) {
        if (parent[to] == i) {
            merge(to, i);
        }
    }
}

for (int i = 0; i < n; ++i) {
    if (revin[i] == -1) {
        continue;
    }
    if (idом[i] == sdом[i]) {
        continue;
    } else {
        idом[i] = idом[idом[i]];
    }
}

vector<int> res(n, -1);
for (int i = 0; i < n; ++i) {
    if (revin[i] == -1) {
        continue;
    }
    res[revin[i]] = revin[idом[i]];
}
return res;
}
};

```

5.4 edmonds_matching.h

```

// O(N^3)
int n;
vi e[maxn];
int mt[maxn], p[maxn], base[maxn], b[maxn], blos[maxn];
int q[maxn];
int blca[maxn]; // used for lca

int lca(int u, int v) {
    for (i = n; blca[i] = 0;
    while (true) {
        u = base[u];
        blca[u] = 1;
        if (mt[u] == -1) break;
        u = p[mt[u]];
    }
    while (!blca[base[v]]) {
        v = p[mt[base[v]]];
    }
    return base[v];
}

void mark_path(int v, int b, int ch) {
    while (base[v] != b) {

```

```

        blos[base[v]] = blos[base[mt[v]]] = 1;
        p[v] = ch;
        ch = mt[v];
        v = p[mt[v]];
    }
}

int find_path(int root) {
    forn(i, n) {
        base[i] = i;
        p[i] = -1;
        b[i] = 0;
    }
    b[root] = 1;
    q[0] = root;
    int lq = 0, rq = 1;
    while (lq != rq) {
        int v = q[lq++];
        for (int to: e[v]) {
            if (base[v] == base[to] || mt[v] == to) ←
                continue;
            if (to == root || (mt[to] != -1 && ←
                p[mt[to]] != -1)) {
                int curbase = lca(v, to);
                forn(i, n) blos[i] = 0;
                mark_path(v, curbase, to);
                mark_path(to, curbase, v);
                forn(i, n) if (blos[base[i]]) {
                    base[i] = curbase;
                    if (!b[i]) b[i] = 1, q[rq++] = i;
                }
            } else if (p[to] == -1) {
                p[to] = v;
                if (mt[to] == -1) {
                    return to;
                }
                to = mt[to];
                b[to] = 1;
                q[rq++] = to;
            }
        }
    }
    return -1;
}

int matching() {
    forn(i, n) mt[i] = -1;
    int res = 0;
    forn(i, n) if (mt[i] == -1) {
        int v = find_path(i);
        if (v != -1) {
            ++res;
            while (v != -1) {
                int pv = p[v], ppv = mt[p[v]];
                mt[v] = pv, mt[ppv] = v;
                v = ppv;
            }
        }
    }
    return res;
}

```

5.5 euler_cycle.h

```

struct Edge {
    int to, id;
};

bool usedEdge[maxn];
vector<Edge> g[maxn];
int ptr[maxn];

vector<int> cycle;
void eulerCycle(int u) {
    while (ptr[u] < sz(g[u]) && ←
        usedEdge[g[u][ptr[u]].id]) ←
        ++ptr[u];
    if (ptr[u] == sz(g[u]))
        return;
    const Edge &e = g[u][ptr[u]];
    usedEdge[e.id] = true;
    eulerCycle(e.to);
}

```

```

    cycle.push_back(e.id);
    eulerCycle(u);
}

```

6 maths

6.1 berlekamp.h

```

vector<int> massey(vector<int> dp) {
    //dp.erase(dp.begin(), dp.begin() + 1);
    vector<int> C(1, 1);
    int L = 0;
    vector<int> B(1, 1);
    int b = 1;
    for (int n = 0; n < dp.size(); ++n) {
        int d = 0;
        for (int i = 0; i <= L; ++i) {
            d += C[i] * dp[n - i];
            d %= mod;
            if (d < 0) {
                d += mod;
            }
        }
        B.insert(B.begin(), 0);
        if (d == 0) {
            continue;
        }
        auto prevC = C;
        if (C.size() < B.size()) {
            C.resize(B.size(), 0);
        }
        int cur_mult = d * binpow(b, mod - 2) % mod;
        for (int i = 0; i < B.size(); ++i) {
            C[i] -= B[i] * cur_mult;
            C[i] %= mod;
            if (C[i] < 0) {
                C[i] += mod;
            }
        }
        if (2 * L <= n) {
            b = d;
            L = n - L + 1;
            B = prevC;
        }
    }
    return C;
}

```

6.2 crt.h

```

inline int inv(int a, int b) {
    return a == 1 ? 1 : b - 1ll * inv(b % a, a) * b ←
        / a % b;
}

pair<int, int> euc(int a, int b) {
    // returns {x, y} s.t. ax + by = g
    int g = __gcd(a, b);
    a /= g, b /= g;
    int x = inv(a, b);
    int y = (1 - 1ll * a * x) / b;

    return {x, y};
}

// be careful if the whole base is long long
pair<int, int> crt(const vector<int>& mods, ←
    vector<int>& rems) {
    int rem = 0, mod = 1;
    for (int i = 0; i < (int)mods.size(); ++i) {
        long long g = __gcd(mods[i], mod);
        if (rem % g != rems[i] % g) {
            return {-1, -1};
        }
    }
    int k = euc(mod, mods[i]).first * 1ll * ←
        (rems[i] - rem + mods[i]) % mods[i];
    if (k < 0) {
        k += mods[i];
    }
    rem += mod / g * k;
    mod = mod / g * mods[i];
}

```

```

    }
    return {rem, mod};
}

```

6.3 gauss_bitset_inverse.h

```

const int N = 100;
using Bs = bitset<N>;
using Matrix = vector<Bs>;

Matrix getInverse(Matrix a) {
    assert(!a.empty());
    int n = a.size();

    Matrix b(n);
    for (int i = 0; i < n; ++i) {
        b[i][i] = 1;
    }

    int row = 0;
    for (int col = 0; col < n; ++col) {
        if (!a[row][col]) {
            int i = row + 1;
            while (i < n && !a[i][col]) {
                ++i;
            }
            if (i == n) {
                return {}; // assert(false);
                ↪ throw PoshelNahuiException();
                ↪ etc
            }
            swap(a[i], a[row]);
            swap(b[i], b[row]);
        }

        for (int i = row + 1; i < n; ++i) {
            if (a[i][col]) {
                a[i] ^= a[row];
                b[i] ^= b[row];
            }
        }

        ++row;
    }

    for (int i = n - 1; i >= 0; --i) {
        for (int j = 0; j < i; ++j) {
            if (a[j][i]) {
                a[j] ^= a[i];
                b[j] ^= b[i];
            }
        }
    }

    return b;
}

```

6.4 gauss_bitset_solve_slu.h

```

const int N = 100;
using Bs = bitset<N>;
using Matrix = vector<Bs>;

Bs solveLinearSystem(Matrix a, Bs b) {
    // solves Av = b
    assert(!a.empty());
    int n = a.size();

    int row = 0;
    vector<int> cols(n);
    for (int col = 0; col < N; ++col) {
        if (row == n) {
            break;
        }
        if (!a[row][col]) {
            int i = row + 1;
            while (i < n && !a[i][col]) {
                ++i;
            }
            if (i == n) {
                continue;
            }
            swap(a[i], a[row]);

```

```

        b[i] = b[i] ^ b[row];
        b[row] = b[row] ^ b[i];
        b[i] = b[i] ^ b[row];
    }

    for (int i = row + 1; i < n; ++i) {
        if (a[i][col]) {
            a[i] ^= a[row];
            b[i] = b[i] ^ b[row];
        }
    }

    cols[row] = col;
    ++row;
}

for (int i = row; i < n; ++i) {
    if (b[i]) {
        return {}; // assert(false); throw
        ↪ PoshelNahuiException(); etc
    }
}

Bs result = {};
while (row) {
    --row;
    for (int i = cols[row] + 1; i < N; ++i) {
        b[row] = b[row] ^ (a[row][i] * result[i]);
    }
    result[cols[row]] = b[row];
}

return result;
}

```

6.5 gauss_double_inverse.h

```

using Matrix = vector<vector<ld>>>;

const ld eps = 1e-6;

Matrix getInverse(Matrix a) {
    assert(!a.empty());
    int n = a.size();
    assert(n == (int)a[0].size());

    Matrix b(n, vector<ld>(n, 0));
    for (int i = 0; i < n; ++i) {
        b[i][i] = 1;
    }

    int row = 0;
    for (int col = 0; col < n; ++col) {
        if (abs(a[row][col]) < eps) {
            int i = row + 1;
            while (i < n && abs(a[i][col]) < eps) {
                ++i;
            }
            if (i == n) {
                return {}; // assert(false);
                ↪ throw PoshelNahuiException();
                ↪ etc
            }
            a[i].swap(a[row]);
            b[i].swap(b[row]);
        }

        for (int i = row + 1; i < n; ++i) {
            ld k = a[i][col] / a[row][col];
            for (int j = col; j < n; ++j) {
                a[i][j] -= k * a[row][j];
            }
            for (int j = 0; j < n; ++j) {
                b[i][j] -= k * b[row][j];
            }
        }

        ++row;
    }

    for (int i = n - 1; i >= 0; --i) {
        for (int j = 0; j < i; ++j) {
            ld k = a[j][i] / a[i][i];

```

```

        for (int l = 0; l < n; ++l) {
            a[j][l] -= a[i][l] * k;
            b[j][l] -= b[i][l] * k;
        }
    }
    ld k = a[i][i];
    for (int l = 0; l < n; ++l) {
        b[i][l] /= k;
    }
    a[i][i] /= k;
}

return b;
}

```

6.6 gauss_double_solve_sluh.h

```

using Matrix = vector<vector<ld>>;

const ld eps = 1e-6;

vector<ld> solveLinearSystem(Matrix a, vector<ld> b) {
    // solves Av = b
    assert(!a.empty());
    int n = a.size(), m = a[0].size();
    assert(n == (int)b.size());

    int row = 0;
    vector<int> cols(n);
    for (int col = 0; col < m; ++col) {
        if (row == n) {
            break;
        }
        if (abs(a[row][col]) < eps) {
            int i = row + 1;
            while (i < n && abs(a[i][col]) < eps) {
                ++i;
            }
            if (i == n) {
                continue;
            }
            a[i].swap(a[row]);
            swap(b[i], b[row]);
        }

        for (int i = row + 1; i < n; ++i) {
            ld k = a[i][col] / a[row][col];
            for (int j = col; j < m; ++j) {
                a[i][j] -= k * a[row][j];
            }
            b[i] -= b[row] * k;
        }

        cols[row] = col;
        ++row;
    }

    for (int i = row; i < n; ++i) {
        if (abs(b[i]) < eps) {
            return {}; // assert(false); throw ←
        }
    }

    vector<ld> result(m);
    while (row) {
        --row;
        for (int i = cols[row] + 1; i < m; ++i) {
            b[row] -= a[row][i] * result[i];
        }
        result[cols[row]] = b[row] / a[row][cols[row]];
    }

    return result;
}

```

6.7 miller_rabin_test.h

```

bool millerRabinTest(ll n, ll a) {
    if (gcd(n, a) > 1)
        return false;
    ll x = n - 1;
    int l = 0;

```

```

    while (x % 2 == 0) {
        x /= 2;
        ++l;
    }
    ll c = binpow(a, x, n);
    for (int i = 0; i < l; ++i) {
        ll nx = mul(c, c, n);
        if (nx == 1) {
            if (c != 1 && c != n - 1)
                return false;
            else
                return true;
        }
        c = nx;
    }
    return c == 1;
}

```

7 misc

7.1 ch_trick_with_binary_summation_struct.cpp

```

const int INF = (int)1e6;

struct Line {
    int k;
    li b;
    bool operator < (const Line& ot) const {
        if (k != ot.k) {
            return k > ot.k;
        }
        return b < ot.b;
    }
    li eval(li x) {
        return k * 1LL * x + b;
    }
};

double get_intersect(Line& q, Line& w) {
    return (q.b - w.b) / 1.0 / (w.k - q.k);
}

struct Hull {
    vector<Line> lines;
    vector<double> borders;
    int Size = 0;
    void append(Line cur) {
        lines.push_back(cur);
    }
    void set_size(int val) {
        Size = val;
    }
    void build() {
        sort(all(lines));
        borders.clear();
        vector<Line> new_lines;
        for (auto& line : lines) {
            if (!new_lines.empty() && new_lines.back().k ←
                == line.k) {
                continue;
            }
            while (new_lines.size() > 1 && ←
                get_intersect(new_lines[new_lines.size() - ←
                2], new_lines.back()) > ←
                get_intersect(new_lines.back(), line)) {
                new_lines.pop_back();
                borders.pop_back();
            }
            if (new_lines.empty()) {
                borders.push_back(-INF);
            } else {
                borders.push_back(get_intersect(new_lines.back(), ←
                line));
            }
            new_lines.push_back(line);
        }
        new_lines.swap(lines);
    }
    int size() {
        return Size;
    }
    li get_min(li x) {

```

```

int id = (int)(lower_bound(all(borders),
    ↪ (double)x) - borders.begin());
li res = (li)1e18;
for (int i = max(id - 1, 0); i < min(id + 2,
    ↪ (int)lines.size()); ++i) {
    res = min(res, lines[i].eval(x));
}
return res;
}
};

struct Lupa {
    vector<Hull> hulls;
    int Size = 0;
    void append_line(Line cur) {
        hulls.push_back(Hull());
        hulls.back().append(cur);
        hulls.back().set_size(1);
        while (hulls.size() >= 2 && hulls.back().size()
            ↪ == hulls[hulls.size() - 2].size()) {
            for (auto& item : hulls.back().lines) {
                hulls[hulls.size() - 2].append(item);
            }
            hulls.pop_back();
            hulls.back().set_size(hulls.back().size() * 2);
        }
        hulls.back().build();
        ++Size;
    }
    li get_min(li x) {
        li res = (li)1e18;
        for (auto& vec : hulls) {
            res = min(res, vec.get_min(x));
        }
        return res;
    }
    int size() {
        return Size;
    }
    void merge_with(Lupa& ot) {
        for (auto& vec : ot.hulls) {
            for (auto& item : vec.lines) {
                append_line(item);
            }
            vec.lines.clear();
        }
    }
    void make_swap(Lupa& ot) {
        swap(ot.Size, Size);
        ot.hulls.swap(hulls);
    }
};

```

7.2 cht_stl.cpp

```

const li is_query = -(1LL << 62);

struct Line {
    // mx + b
    li m, b;
    mutable function<const Line *(> succ;

    bool operator<(const Line &rhs) const {
        if (rhs.b != is_query) return m < rhs.m;
        const Line *s = succ();
        if (!s) return 0;
        li x = rhs.m;
        return b - s->b < (s->m - m) * x;
    }
};

using LI = __int128_t; // or long double; long long
    ↪ if line coords are <= 1e9

// WARNING: don't try to swap this structure (e.g.
    ↪ in lower to greater):
// it will make next iterators inconsistent and SIGSEGV
struct HullDynamic : public multiset<Line> {
    bool bad(iterator y) {
        auto z = next(y);
        if (y == begin()) {
            if (z == end()) return 0;
            return y->m == z->m && y->b <= z->b;
        }
    }
};

```

```

}
auto x = prev(y);
if (z == end()) return y->m == x->m && y->b
    ↪ <= x->b;

return (x->b - y->b) * (LI)(z->m - y->m) >=
    ↪ (y->b - z->b) * (LI)(y->m - x->m);
}

void insert_line(li m, li b) {
    auto y = insert({m, b});
    y->succ = [=] { return next(y) == end() ? 0
        ↪ : &*next(y); };
    if (bad(y)) {
        erase(y);
        return;
    }
    while (next(y) != end() && bad(next(y)))
        ↪ erase(next(y));
    while (y != begin() && bad(prev(y)))
        ↪ erase(prev(y));
}

li getMax(li x) {
    auto l = *lower_bound((Line) {x, is_query});
    return l.m * x + l.b;
}
};

```

7.3 tree_bidirectional_dp.h

```

/* For any commutative function f({x, y, ..., z}) =
    ↪ f(x, f(y, f(..., z)))
   * like sum, min, max, or, xor, and, etc
   * calculates in dp[i][j] f(subtree),
   * where subtree is a connectivity component of G \
    ↪ (i, a[i][j]) with vertex a[i][j]
   */

const int N = 222222;
vector<int> a[N];
vector<int> dp[N];
int par[N];

#define data asdf
int data[N];

inline int f(int x, int y) {
    return x | y;
}

int dfsDown(int v) {
    int res = data[v];
    for (int i = 0; i < (int)a[v].size(); ++i) {
        int to = a[v][i];
        if (to == par[v]) {
            continue;
        }
        par[to] = v;
        res = f(res, dp[v][i] = dfsDown(to));
    }
    return res;
}

void dfsUp(int v, int to_parent = 0) {
    vector<int> pref, suf;
    pref.reserve(a[v].size());
    suf.reserve(a[v].size());
    int j = 0;
    for (int i = 0; i < (int)a[v].size(); ++i) {
        int to = a[v][i];
        if (to == par[v]) {
            dp[v][i] = to_parent;
            continue;
        }
        pref.push_back(j ? f(pref[j - 1], dp[v][i])
            ↪ : dp[v][i]);
        ++j;
    }
    j = 0;
    for (int i = (int)a[v].size() - 1; i >= 0; --i) {
        int to = a[v][i];
        if (to == par[v]) {

```

```

        continue;
    }
    suf.push_back(j ? f(dp[v][i], suf[j - 1]) : ←
    ↪ dp[v][i]);
    ++j;
}
reverse(all(suf));

j = 0;
to_parent = f(to_parent, data[v]);
for (int i = 0; i < (int)a[v].size(); ++i) {
    int to = a[v][i];
    if (to == par[v]) {
        continue;
    }
    int new_to_parent = to_parent;
    if (j > 0) {
        new_to_parent = f(pref[j - 1], ←
        ↪ new_to_parent);
    }
    if (j < (int)suf.size() - 1) {
        new_to_parent = f(new_to_parent, suf[j + ←
        ↪ 1]);
    }
    dfsUp(to, new_to_parent);
    ++j;
}
}
}

```

7.4 tree_order_statistics.cpp

```

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <bits/stdc++.h>

using namespace std;
using namespace __gnu_pbds;

using orderedSet = tree<
    int,
    null_type,
    less<int>,
    rb_tree_tag,
    tree_order_statistics_node_update
>;

int main() {
    orderedSet X;
    X.insert(1);
    X.insert(2);
    X.insert(4);
    X.insert(8);
    X.insert(16);

    std::cout << *X.find_by_order(1) << std::endl; // 2
    std::cout << *X.find_by_order(2) << std::endl; // 4
    std::cout << *X.find_by_order(4) << std::endl; // 16
    std::cout << std::boolalpha <<
        (end(X) == X.find_by_order(6)) << std::endl; ←
    ↪ // true

    std::cout << X.order_of_key(-5) << std::endl; // 0
    std::cout << X.order_of_key(1) << std::endl; // 0
    std::cout << X.order_of_key(3) << std::endl; // 2
    std::cout << X.order_of_key(4) << std::endl; // 2
    std::cout << X.order_of_key(400) << std::endl; // 5
}

```

8 numeric

8.1 integration.cpp

```

template<typename F>
F integrate(F (*f)(F), F a, F b, int nodes){
    F d = (b - a)/(nodes + 1);
    F ans = 0;
    for(int i = 0; i < nodes + 1; i++){
        F L = a, R = a + d;
        ans += d*(f(L) + f(R) + 4*f(0.5 * (L + R)))/6;
        a = R;
    }
}

```

```

    return ans;
}

```

8.2 simplex.cpp

```

//indexes
//0: constant
//1..N: non-basic variables
//N+1..B+N+1: basic variables
template<typename F>
class CanonicalSolver{
public:
    static F* solve_feasible(int B, int N, int * lhs,
        F ** rhs, F * func, F eps){
        F * values = new F[B + N + 1];
        memset(values, 0, sizeof(F) * (B + N + 1));
        for(int i = 0; i < B; i++){
            values[lhs[i]] = rhs[i][0];
            values[0] = 1;
            bool * basis = new bool[B + N + 1];
            memset(basis, 0, sizeof(bool) * (B + N + 1));
            while(1){
                int pos = -1;
                for(int i = 0; i < B; i++){
                    basis[lhs[i]] = 1;
                }
                for(int i = 1; i < B + N + 1; i++){
                    if(basis[i] || func[i] < eps)
                        continue;
                    if(pos == -1 || func[i] > func[pos])
                        pos = i;
                }
                for(int i = 0; i < B; i++){
                    basis[lhs[i]] = 0;
                }
                if(pos == -1)break;
                F bnd = 0;
                bool was = 0;
                int what = 0;
                for(int i = 0; i < B; i++){
                    if(rhs[i][pos] > -eps)
                        continue;
                    F curr = values[lhs[i]];
                    curr /= -rhs[i][pos];
                    if(!was || bnd > curr){
                        was = 1;
                        what = i;
                        bnd = curr;
                    }
                }
                if(!was)
                    return nullptr;
                for(int i = 0; i < B; i++){
                    values[lhs[i]] += bnd * rhs[i][pos];
                }
                int old = lhs[what];
                lhs[what] = pos;
                values[pos] += bnd;
                F oldval = 1/rhs[what][pos];
                for(int i = 0; i < 1 + B + N; i++){
                    rhs[what][i] *= -oldval;
                }
                rhs[what][old] = oldval;
                rhs[what][pos] = 0;
                for(int i = 0; i < B; i++){
                    if(i == what)
                        continue;
                    F coeff = rhs[i][pos];
                    rhs[i][pos] = 0;
                    for(int j = 0; j < 1 + B + N; j++){
                        rhs[i][j] += rhs[what][j] * coeff;
                    }
                }
                F coeff = func[pos];
                func[pos] = 0;
                for(int j = 0; j < 1 + B + N; j++){
                    func[j] += rhs[what][j] * coeff;
                }
            }
            delete[] basis;
            return values;
        }
        //0: solution exists
        //1: unbounded
        //-1: unfeasible
        static pair<F*, int> solve(int B, int N, int * lhs,
            F ** rhs, F * func, F eps){
            bool fea = 1;
            for(int i = 0; i < B; i++){
                if(rhs[i][0] < -eps){fea = 0; break;}
            }
        }
    }
}

```



```

if(fea){
    auto res = solve_feasible(B, N, lhs, rhs,
        func, eps);
    return res == nullptr ? make_pair(res, 1) :
        make_pair(res, 0);
}
int pos = 0;
for(int i = 1; i < B; i++){
    if(rhs[i][0] < rhs[pos][0])
        pos = i;
    int * new_lhs = new int[B];
    memcpy(new_lhs, lhs, B * sizeof(int));
    F ** new_rhs = (F**)malloc(B * sizeof(F));
    for(int i = 0; i < B; i++){
        new_rhs[i] = (F*)malloc((2 + B + N) *
            sizeof(F));
        memcpy(new_rhs[i], rhs[i], (1 + B + N) *
            sizeof(F));
        new_rhs[i][1 + B + N] = 1;
    }
    F * new_func = new F[2 + N + B];
    memset(new_func, 0, sizeof(F) * (2 + N + B));
    new_rhs[pos][1 + N + B] = 0;
    for(int j = 0; j < 2 + N + B; j++){
        new_rhs[pos][j] = -new_rhs[pos][j];
    }
    new_rhs[pos][lhs[pos]] = 1;
    new_lhs[pos] = 1 + N + B;
    for(int i = 0; i < B; i++){
        if(pos == i)
            continue;
        new_rhs[i][1 + N + B] = 0;
        for(int j = 0; j < 1 + N + B; j++){
            new_rhs[i][j] += new_rhs[pos][j];
        }
    }
    for(int i = 0; i < 1 + N + B; i++){
        new_func[i] = -new_rhs[pos][i];
    }
    auto res_lambda = solve_feasible(B, N + 1,
        new_lhs,
        new_rhs, new_func, eps);
    if(res_lambda == nullptr)
        return make_pair(nullptr, -1);
    F cres = 0;
    for(int i = 0; i < 2 + N + B; i++){
        cres += res_lambda[i] * new_func[i];
    }
    if(abs(cres) > eps)
        return make_pair(nullptr, -1);
    int bpos = -1;
    for(int i = 0; i < B; i++){
        if(new_lhs[i] == 1 + N + B){
            bpos = i;
            break;
        }
    }
    if(bpos == -1){
        memcpy(lhs, new_lhs, B * sizeof(int));
        for(int i = 0; i < B; i++){
            memcpy(rhs[i], new_rhs[i], (1 + B + N) *
                sizeof(F));
        }
        memcpy(new_func, func, (1 + B + N) *
            sizeof(F));
        for(int i = 0; i < B; i++){
            F coeff = func[new_lhs[i]];
            new_func[new_lhs[i]] = 0;
            for(int j = 0; j < 1 + B + N; j++){
                new_func[j] += coeff *
                    new_rhs[i][j];
            }
        }
        memcpy(func, new_func, (1 + B + N) *
            sizeof(F));
        auto res = solve_feasible(B, N, lhs, rhs,
            func, eps);
        return res == nullptr ? make_pair(res, 1) :
            make_pair(res, 0);
    }
    int with_what = -1;
    for(int i = 1; i < 1 + N + B; i++){
        if(abs(new_rhs[bpos][i]) > eps){
            with_what = i;
            break;
        }
    }
    F coeff = -new_rhs[bpos][with_what];
    new_rhs[bpos][with_what] = 0;
    new_rhs[bpos][new_lhs[bpos]] = -1;
    new_lhs[bpos] = with_what;

```

```

for(int j = 0; j < 2 + N + B; j++){
    new_rhs[bpos][j] /= coeff;
}
for(int i = 0; i < B; i++){
    if(i == bpos)
        continue;
    F coeff = new_rhs[i][with_what];
    for(int j = 0; j < 2 + N + B; j++){
        new_rhs[i][j] += coeff *
            new_rhs[bpos][j];
    }
}
memcpy(lhs, new_lhs, B * sizeof(int));
for(int i = 0; i < B; i++){
    memcpy(rhs[i], new_rhs[i], (1 + B + N) *
        sizeof(F));
}
memcpy(new_func, func, (1 + B + N) * sizeof(F));
for(int i = 0; i < B; i++){
    F coeff = func[new_lhs[i]];
    new_func[new_lhs[i]] = 0;
    for(int j = 0; j < 1 + B + N; j++){
        new_func[j] += coeff * new_rhs[i][j];
    }
}
memcpy(func, new_func, (1 + B + N) * sizeof(F));
auto res = solve_feasible(B, N, lhs, rhs,
    func, eps);
return res == nullptr ? make_pair(res, 1) :
    make_pair(res, 0);
}

```

};

9 strings

9.1 aho_corasick.h

```

const int ALPHABET = 26;

struct state {
    array<int, ALPHABET> transition = {};
    int link = 0;

    bool isTerminal = false;
};

struct automaton {
    vector<state> states = { state() };
    int numStates = 1;

    void addString(const string& s) {
        int cur = 0;
        for (char c: s) {
            c -= 'a';
            int& to = states[cur].transition[c];
            if (to) {
                cur = to;
            }
            else {
                cur = to = states.size();
                states.push_back(state());
            }
        }
        states[cur].isTerminal = true;
    }

    void build() {
        deque<int> q;
        q.push_back(0);

        while (!q.empty()) {
            int v = q.front();
            q.pop_front();
            states[v].isTerminal =
                states[v].isTerminal ||
                states[states[v].link].isTerminal;

            for (int c = 0; c < ALPHABET; ++c) {
                if (int u = states[v].transition[c]) {
                    states[u].link = v ?
                        states[states[v].link].transition[c]
                        : 0;
                    q.push_back(u);
                }
                else {
                    states[v].transition[c] =
                        states[states[v].link].transition[c];
                }
            }
        }
    }
}

```

```

    }
    }
}
};

```

9.2 manacher.h

```

array<vector<int>, 2> manacher(const string& s) {
    int n = s.length();
    array<vector<int>, 2> res;
    for (auto& v : res) {
        v.assign(n, 0);
    }
    for (int z = 0, l = 0, r = 0; z < 2; ++z, l = 0, ←
    ↪ r = 0) {
        for (int i = 0; i < n; ++i) {
            if (i < r) {
                res[z][i] = min(r - i + !z, res[z][l ←
                ↪ + r - i + !z]);
            }
            int L = i - res[z][i], R = i + res[z][i] ←
            ↪ - !z;
            while (L - 1 >= 0 && R + 1 < n && s[L - ←
            ↪ 1] == s[R + 1]) {
                ++res[z][i];
                --L;
                ++R;
            }
            if (R > r) {
                l = L;
                r = R;
            }
        }
    }
    return res;
}

```

9.3 pal_tree.h

```

struct node{
    map<char, int> nxt;
    int link;
    int len;
    int diff;
    int series;
};

node t[N];
int sz;
int last;

void init(){
    sz = 2;
    t[0].link = 1;
    t[0].len = 0;
    t[1].len = t[1].link = -1;
    last = 1;
}

void extend(const string & s, int pos){
    int v = last;
    while(1){
        if(s[pos] != s[pos - 1 - t[v].len])
            v = t[v].link;
        else break;
    }
    if(t[v].nxt.count(s[pos])){
        v = t[v].nxt[s[pos]];
    }
    else{
        int u = sz++;
        t[v].nxt[s[pos]] = u;
        t[u].len = t[v].len + 2;
        while(v != -1){
            v = t[v].link;
            if(s[pos] == s[pos - 1 - t[v].len])
                break;
        }
        if(v == -1){
            t[u].link = 0;
        }
        else t[u].link = t[v].nxt[s[pos]];
    }
}

```

```

    t[u].diff = t[u].len - t[t[u].link].len;
    if(t[u].diff == t[t[u].link].diff)
        t[u].series = t[t[u].link].series;
    else
        t[u].series = t[u].link;
    v = u;
}
last = v;
}

```

9.4 prefix_function.h

```

void prefixFunction(const string& s, vector<int>& p) {
    if (s.length() == 0)
        return;
    p[0] = 0;
    for (size_t i = 1; i < s.length(); ++i) {
        int j = p[i - 1];
        while (j > 0 && s[i] != s[j])
            j = p[j - 1];
        if (s[i] == s[j])
            ++j;
        p[i] = j;
    }

    const char first = 'a';
    const int alphabet = 26;
    // вылезит из массива, после того, как совпадет все. ←
    ↪ можно добавить aut[n] = aut[p[n - 1]]
    // это эмулирует переход по суф ссылке
    vector<vi> pfautomaton(const string& s) {
        vi p(s.length());
        prefixFunction(s, p);
        vector<vi> aut(s.length(), vi(alphabet));
        for (size_t i = 0; i < s.length(); ++i) {
            for (char c = 0; c < alphabet; ++c) {
                if (i > 0 && c != s[i] - first) {
                    aut[i][c] = aut[p[i - 1]][c];
                }
                else {
                    aut[i][c] = i + (c == s[i] - first);
                }
            }
        }
        return aut;
    }
}

```

9.5 suffix_array.cpp

```

void Build(const string& init, vector<int>& ←
    ↪ suffArray, vector<int>& lcp) {
    string s = init;
    s.push_back(char(0));
    int n = s.size();
    vector<int> head(max(n, 256));
    vector<int> color(n);
    vector<int> colorSub(n);
    vector<int> suffArraySub(n);
    lcp.resize(n);
    suffArray.resize(n);

    for (int i = 0; i < s.size(); ++i) {
        ++head[s[i]];
    }
    for (int i = 1; i < 256; ++i) {
        head[i] += head[i - 1];
    }
    for (int i = 255; i > 0; --i) {
        head[i] = head[i - 1];
    }
    head[0] = 0;
    for (int i = 0; i < s.size(); ++i) {
        suffArray[head[s[i]]] = i;
        ++head[s[i]];
    }
    int numClasses = 1;
    head[0] = 0;
    for (int i = 1; i < s.size(); ++i) {
        if (s[suffArray[i - 1]] != s[suffArray[i]]) {
            ++numClasses;
            head[numClasses - 1] = i;
        }
    }
}

```

```

    color[suffArray[i]] = numClasses - 1;
}
for (int k = 1; k < s.size(); k *= 2) {
    for (int i = 0; i < s.size(); ++i) {
        int first = suffArray[i] - k;
        if (first < 0) {
            first += s.size();
        }
        suffArraySub[head[color[first]]] = first;
        ++head[color[first]];
    }
    suffArray = suffArraySub;

    int second;
    pair<int, int> prevClasses, curClasses;
    curClasses = { -1, 0 };
    numClasses = 0;

    for (int i = 0; i < s.size(); ++i) {
        prevClasses = curClasses;

        second = suffArray[i] + k;
        if (second >= s.size()) {
            second -= s.size();
        }
        curClasses = { color[suffArray[i]],
            color[second] };

        if (curClasses != prevClasses) {
            ++numClasses;
            head[numClasses - 1] = i;
        }
        colorSub[suffArray[i]] = numClasses - 1;
    }

    color = colorSub;

    if (numClasses == s.size())
        break;
}
vector<int> pos;
int curLcp = 0;
pos.resize(s.size());
for (int i = 0; i < s.size(); ++i) {
    pos[suffArray[i]] = i;
}
lcp.resize(s.size());
for (int i = 0; i < s.size(); ++i) {
    if (pos[i] == s.size() - 1) {
        lcp[pos[i]] = 0;
        curLcp = 0;
        continue;
    }

    while (s[(i + curLcp) % s.size()] ==
        s[(suffArray[pos[i] + 1] + curLcp) %
        s.size()]) {
        ++curLcp;
    }
    lcp[pos[i]] = curLcp;

    --curLcp;
    if (curLcp < 0)
        curLcp = 0;
}

void BuildSparseTable(const vector<int>& a, vector<
    vector<int>>& sparseTable) {
    int logSize = 0;
    while ((1 << logSize) < a.size()) {
        ++logSize;
    }
    logSize = 19; // <-- THINK HERE!
    sparseTable.assign(a.size(), vector<int>
        (logSize + 1));

    for (int i = 0; i < a.size(); ++i) {
        sparseTable[i][0] = a[i];
    }

    for (int k = 1; k <= logSize; ++k) {
        for (int i = 0; i + (1 << k) <= a.size(); ++i) {

```

```

            sparseTable[i][k] = min(sparseTable[i][k - 1],
                sparseTable[i + (1 << (k - 1))][k - 1]);
        }
    }

    int GetMin(int l, int r, const vector<vector<int>>
        & sparseTable) {
        assert(l < r);
        int sz = 31 - __builtin_clz(r - l);
        return min(sparseTable[l][sz], sparseTable[r -
            (1 << sz)][sz]);
    }

    void solve(__attribute__((unused)) bool read) {
        string s;
        cin >> s;
        int n = s.length();
        vector<int> suffArray, lcp;
        Build(s, suffArray, lcp);
        suffArray.erase(suffArray.begin());
        lcp.erase(lcp.begin());
        vector<int> pos_in_array(n);
        for (int i = 0; i < suffArray.size(); ++i) {
            pos_in_array[suffArray[i]] = i;
        }
        vector<vector<int>> sparse;
        BuildSparseTable(lcp, sparse);
    }
}

9.6 suffix_automaton_kostroma.h

const int UNDEFINED_VALUE = -1;

class SuffixAutomaton {
public:
    struct State {
        map<char, int> transitions;
        int link;
        int maxlen;
        int firstPos, lastPos;
        int cnt;
        State():link(UNDEFINED_VALUE),
            firstPos(UNDEFINED_VALUE),
            lastPos(UNDEFINED_VALUE), maxlen(0),
            cnt(0) {}
    };
    vector<State> states;
    int lastState;
    SuffixAutomaton(const string& s) {
        states.push_back(State());
        lastState = 0;
        for (int i = 0; i < s.length(); ++i)
            append(s[i]);
        vector<pair<int, int>> p(states.size());
        for (int i = 0; i < p.size(); ++i) {
            p[i].second = i;
            p[i].first = states[i].maxLen;
        }
        sort(all(p));
        reverse(all(p));
        for (int i = 0; i < p.size(); ++i) {
            int curState = p[i].second;
            if (states[curState].lastPos ==
                UNDEFINED_VALUE)
                states[curState].lastPos =
                    states[curState].firstPos;
            if (states[curState].link !=
                UNDEFINED_VALUE) {
                states[states[curState].link].lastPos =
                    max(states[states[curState].link].lastPos,
                    states[curState].lastPos);
                states[states[curState].link].cnt +=
                    states[curState].cnt;
            }
        }
    }

private:

```

```

void append(char c) {
    int curState = states.size();
    states.push_back(State());
    states[curState].maxLen =
        ↪ states[lastState].maxLen + 1;
    states[curState].firstPos =
        ↪ states[lastState].maxLen;
    states[curState].cnt = 1;
    int prevState = lastState;
    for (; prevState != UNDEFINED_VALUE;
        ↪ prevState = states[prevState].link) {
        if (states[prevState].transitions.count(c))
            break;
        states[prevState].transitions[c] = curState;
    }

    if (prevState == UNDEFINED_VALUE) {
        states[curState].link = 0;
    }
    else {
        int nextState =
            ↪ states[prevState].transitions[c];
        if (states[nextState].maxLen ==
            ↪ states[prevState].maxLen + 1) {
            states[curState].link = nextState;
        }
        else {
            int cloneState = states.size();
            states.push_back(State());
            states[cloneState].maxLen =
                ↪ states[prevState].maxLen + 1;
            states[cloneState].link =
                ↪ states[nextState].link;
            states[cloneState].firstPos =
                ↪ states[nextState].firstPos;
            states[curState].link =
                ↪ states[nextState].link =
                ↪ cloneState;

            states[cloneState].transitions =
                ↪ states[nextState].transitions;
            for (; prevState != UNDEFINED_VALUE
                ↪ &&
                ↪ states[prevState].transitions[c]
                ↪ == nextState; prevState =
                ↪ states[prevState].link)
                ↪ states[prevState].transitions[c]
                ↪ = cloneState;
        }
    }
    lastState = curState;
}
};

```

9.7 suffix_tree_from_automaton.cpp

```

struct SuffixTree {
    vector<vector<pair<int, int>>> g;
    vector<int> is_leaf, max_len;
    vector<int> leaves_before;
    vector<int> cnt_leaves;
    int n;
    SuffixTree(vector<int> s) {
        s.push_back(-1);
        reverse(all(s));
        n = s.size();
        auto automata = SuffixAutomaton(s);
        g.resize(automata.states.size());
        is_leaf.resize(automata.states.size(), 0);
        max_len.assign(g.size(), 0);
        cnt_leaves.assign(g.size(), 0);
        leaves_before.assign(g.size(), 0);
        for (int v = 1; v < automata.states.size(); ++v) {
            int p = automata.states[v].link;
            max_len[v] = automata.states[v].maxLen;
            is_leaf[v] = automata.states[v].firstPos + 1
                ↪ == automata.states[v].maxLen;
            int transition_pos =
                ↪ automata.states[v].lastPos -
                ↪ automata.states[p].maxLen;
            g[p].push_back({s[transition_pos], v});
        }
    }
};

```

```

    }
    for (auto& vec : g) {
        sort(all(vec));
    }
    vector<int> new_leaves;
    for (int i = 0; i < g.size(); ++i) {
        vector<int> to_erase;
        for (int j = 0; j < g[i].size(); ++j) {
            int to = g[i][j].second;
            if (is_leaf[to]) {
                --max_len[to];
                if (max_len[to] == max_len[i]) {
                    to_erase.push_back(j);
                    is_leaf[to] = false;
                    if (i > 0) {
                        new_leaves.push_back(i);
                    }
                }
            }
        }
        vector<pair<int, int>> copy_g;
        int uk = 0;
        for (int j = 0; j < g[i].size(); ++j) {
            if (uk < to_erase.size() && j == to_erase[uk]) {
                ++uk;
                continue;
            }
            copy_g.push_back(g[i][j]);
        }
        copy_g.swap(g[i]);
        for (int v : new_leaves) {
            is_leaf[v] = 1;
        }
    }
};

9.8 z_function.h

vector<int> zFunction(const string& s) {
    int n = s.length();

    vector<int> z(n);
    int l = 0, r = 0;
    for (int i = 1; i < n; ++i) {
        z[i] = max(min(z[i - l], r - i), 0);

        while (i + z[i] < n && s[i + z[i]] == s[z[i]])
            ++z[i];

        if (i + z[i] > r) {
            l = i;
            r = i + z[i];
        }
    }

    if (n)
        z[0] = n;

    return z;
}

```

10 templates

10.1 template.cpp

```

//g++ options: -Wall -Wextra -O2 --std=c++17 -DLOCAL
//#pragma GCC optimize('Ofast,unroll-loops')
//#pragma GCC target('avx2,tune=native')
#include <bits/stdc++.h>

using namespace std;

#define all(v) (v).begin(), (v).end()
#define sz(a) ((ll)(a).size())
#define X first
#define Y second

using ll = long long;
using ull = unsigned long long;
using dbl = long double;
mt19937_64
    ↪ rng(chrono::steady_clock::now().time_since_epoch().count());

```

```

11 myRand(11 mod) {
    return (ull)rng() % mod;
}

void solve() {

}

signed main() {
#ifdef LOCAL
    assert(freopen("input.txt", "r", stdin));
    // assert(freopen("output.txt", "w", stdout));
#endif
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    cout << fixed << setprecision(20);

    int T = 1;
    // cin >> T;
    for (int i = 0; i < T; ++i) {
        solve();
    }

#ifdef LOCAL
    cout << endl << endl << "time = " << clock() /
        (double)CLOCKS_PER_SEC << endl;
#endif
}

```

11 treap

11.1 treap.cpp

// fuckup: don't forget to push in recursive walk

```

int getrand() {
    /*static std::random_device rd;
    static std::mt19937 generator(rd());
    static std::uniform_int_distribution<int>
    distribution(0, INT_MAX);
    return distribution(generator);*/
    return rand() ^ (rand() << 15);
}

struct Node {
    Node *left;
    Node *right;
    int priority;
    int size;
    ll value;
    ll sum;
    ll add;
    bool isReversed;

    explicit Node(ll value): left(nullptr),
        right(nullptr), value(value) {
        priority = getrand();
        size = 1;
        sum = value;
        isReversed = false;
        add = 0;
    }
};

int getSize(Node *node) {
    return node ? node->size: 0;
}

ll getSum(Node *node) {
    return node ? node->sum: 0;
}

void addToNode(Node *node, ll value) {
    if (node) {
        node->value += value;
        node->sum += value * getSize(node);
        node->add += value;
    }
}

void reverseNode(Node *node) {
    if (node) {
        std::swap(node->left, node->right);
    }
}

```

```

        node->isReversed = !node->isReversed;
    }
}

void push(Node *node) {
    if (!node) return;
    if (node->isReversed) {
        reverseNode(node->left);
        reverseNode(node->right);
    }
    if (node->add) {
        addToNode(node->left, node->add);
        addToNode(node->right, node->add);
    }
    node->isReversed = false;
    node->add = 0;
}

void recalc(Node *node) {
    node->size = 1 + getSize(node->left) +
        getSize(node->right);
    node->sum = node->value + getSum(node->left) +
        getSum(node->right);
}

Node* Merge(Node *left, Node *right) {
    if (!right)
        return left;
    if (!left)
        return right;
    push(left);
    push(right);
    if (left->priority > right->priority) {
        left->right = Merge(left->right, right);
        recalc(left);
        return left;
    } else {
        right->left = Merge(left, right->left);
        recalc(right);
        return right;
    }
}

std::pair<Node*, Node*> Split(Node *node, int k) {
    /*return (T1, T2). |T1| = max(0, min(k, |node|))*/
    if (!node)
        return {nullptr, nullptr};
    push(node);
    if (getSize(node->left) < k) {
        Node *left, *right;
        std::tie(left, right) = Split(node->right, k
            - 1 - getSize(node->left));
        node->right = left;
        recalc(node);
        return {node, right};
    } else {
        Node *left, *right;
        std::tie(left, right) = Split(node->left, k);
        node->left = right;
        recalc(node);
        return {left, node};
    }
}

std::pair<Node*, Node*> SplitByValue(Node *node, int
    value) {
    /*use only if tree is sorted*/
    /*return (T1, T2). For all x in T1 x < value*/
    if (!node)
        return {nullptr, nullptr};
    push(node);
    if (node->value < value) {
        Node *left, *right;
        std::tie(left, right) =
            SplitByValue(node->right, value);
        node->right = left;
        recalc(node);
        return {node, right};
    } else {
        Node *left, *right;
        std::tie(left, right) =
            SplitByValue(node->left, value);
        node->left = right;
    }
}

```

```

        recalc(node);
        return {left, node};
    }
}

void Insert(Node* &node, int pos, ll value) {
    Node *left, *right;
    std::tie(left, right) = Split(node, pos);
    node = Merge(Merge(left, new Node(value)), right);
}

void Remove(Node* &node, int pos) {
    Node *left, *mid, *right;
    std::tie(left, right) = Split(node, pos + 1);
    std::tie(left, mid) = Split(left, pos);
    delete mid;
    node = Merge(left, right);
}

template<typename Function>
void queryOnSegment(Node* &node, int l, int r,          ↵
    ↵ Function callback) {
    Node *left, *mid, *right;
    std::tie(left, right) = Split(node, r + 1);
    std::tie(left, mid) = Split(left, l);
    callback(mid);
    node = Merge(Merge(left, mid), right);
}

ll getSumOnSegment(Node* &root, int l, int r) {
    ll answer;
    queryOnSegment(root, l, r, [&answer] (Node*      ↵
    ↵ &node) {answer = getSum(node);});
    return answer;
}

void addToSegment(Node* &root, int l, int r, ll value) {
    queryOnSegment(root, l, r, [value] (Node* &node) ↵
    ↵ {addToNode(node, value);});
}

```